Clube Delphi

Business Intelligence
Análises de dados com FastCube

Dicionários de dados
Simplificando a atualizando o banco de dados

Code Smells, Bad Smells e Clean Code
Detecte e corrija problemas no código fonte

Edição 166 :: R$ 14,90

DEVMEDIA

ISSN 1517990-7

DESENVOLVIMENTO MOBILE NO DELPHI

Crie apps para Android no Delphi 10 Seattle

MVP

R$ 1.000.000,00
INVESTIDOS EM CONTEÚDO NOS ÚLTIMOS 12 MESES.

APLIQUE ESSE INVESTIMENTO NA SUA CARREIRA...

E MOSTRE AO MERCADO QUANTO VOCÊ VALE!

CONFIRA TODO O MATERIAL QUE VOCÊ TERÁ ACESSO:

- + de **9.000** video-aulas
- + de **290** cursos online
- + de **13.000** artigos
- DEVMEDIA API's consumido + de **500.000** vezes

POR APENAS
R$ 69,90* mensais

*Tempo mínimo de assinatura: 12 meses.

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

PRA QUEM QUER EXIGIR
MAIS DO MERCADO!

DEVMEDIA

Clube Delphi

166ª Edição 2016 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Joel Rodrigues (joelrlneto@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultora Técnica

Daniella Costa (daniella.devmedia@gmail.com)

Capa e Diagramação

Romulo Araujo

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Joel Rodrigues - Editor da Revista
joelrlneto@gmail.com

Joel Rodrigues - **Editor Geral**

Editor da ClubeDelphi Magazine

Sumário

Artigo no estilo Solução Completa

04 – Business Intelligence utilizando o FastCube
[*Jones Granatyr, Edson Emílio Scalabrin, Fábio Alexandre Taffe e Alan Diego Darold*]

Artigo no estilo Solução Completa

16 – Construindo aplicativos Android no Delphi 10 Seattle
[*Jones Granatyr, Guinther Pauli, Thiago Luiz Lauxen, Jean Paul Barddal, Fábio Spak e Fábio Alexandre Taffe*]

Conteúdo sobre Boas Práticas

26 – Automatizando a atualização do banco de dados em Delphi
[*Gutierry Antonio*]

Conteúdo sobre Boas Práticas, Artigo no estilo Mentoring

41 – Como corrigir Bad Smells com refatoração em Delphi
[*Guinther Pauli*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Business Intelligence utilizando o FastCube

Aprenda a construir uma aplicação em Delphi para análise de dados

Fique por Dentro

As técnicas de BI (Business Intelligence) estão sendo cada vez mais utilizadas por empresas com o intuito de possibilitar vantagens competitivas, auxiliando em particular nos processos de tomada de decisões estratégicas. Nesse contexto, este artigo é útil para quem deseja oferecer aos seus clientes ferramentas para análise de dados, que poderão funcionar, por exemplo, como indicadores de desempenho, os quais podem mostrar de maneira rápida e visual – por meio de gráficos – como está a situação da empresa em um dado momento no que diz respeito ao cumprimento de metas. Com o FastCube é possível apresentar dados multidimensionais na forma de gráficos intuitivos e simples de serem analisados.

Conhecimento é um componente essencial para qualquer organização. Ele é o elemento chave e valioso para a tomada de uma decisão estratégica segura em uma empresa. É por meio da correta utilização do conhecimento que uma empresa terá melhor condição de decidir sobre, por exemplo, quando realizar o lançamento de um novo produto, para que tipo de mercado o produto deve ser divulgado ou quais são as previsões de lucro dado que uma estratégia específica de *marketing* será executada. Além disso, a disponibilidade de dados – em um esquema multidimensional – sobre situações passadas da empresa possibilita a descoberta de padrões de comportamentos por meio de algoritmos de aprendizagem de máquina. Esses padrões podem possibilitar a realização de previsões futuras de como uma empresa estará nos próximos anos caso continue seguindo os padrões identificados. Desta forma, informações que aparentemente estejam perdidas em meio a milhares de dados podem ser recuperadas.

Em geral, empresas tanto do setor comercial quanto do setor industrial, ainda usam sistemas de informação transacionais para o processamento dos dados de seus negócios. Esses sistemas são assim chamados por tratarem das atividades do cotidiano da empresa. Eles são voltados para operações de manutenção de dados, ou seja, inclusão, alteração, exclusão e consulta. Por exemplo, um software de hotelaria irá gerenciar os principais processos deste setor, bem como a entrada/saída de hóspedes, reserva de apartamentos e controle dos consumos do frigobar. Por outro lado, um software para controle de folha de pagamento terá como principais funções o cadastro dos funcionários, a geração da folha de pagamento e os cálculos dos impostos a pagar. Em geral, esses sistemas possibilitam a visualização de relatórios gerenciais, os quais apresentam informações sobre a operação da empresa. Uma listagem das reservas de apartamentos por data e uma listagem da folha de pagamento do mês são dois exemplos para esses cenários.

Na maioria das vezes, somente este tipo de informação não é suficiente para que a empresa possa efetivamente tomar uma decisão. Responder questões como: Qual o perfil do hóspede que o hotel recebe? São hóspedes que viajam a trabalho ou para turismo? As respostas para essas questões podem gerar conhecimento, ou seja, com base no perfil dos hóspedes podem ser determinadas estratégias de *marketing* para a divulgação do hotel. Neste sentido, pode-se definir se a campanha de divulgação terá como público alvo empresas que possuem vendedores viajantes ou sites de turismo da cidade. Com relação ao exemplo da folha de pagamento, ter o conhecimento de quantos funcionários a empresa poderá manter o salário nos próximos anos pode levar à tomada de decisão sobre a contração ou não de novos funcionários. Como pôde ser observado nesses dois exemplos, o conhecimento possui muito mais profundidade do que a informação, ou seja, enquanto a informação possui um caráter meramente informativo, o conhecimento auxilia nos processos de tomada de decisões estratégicas.

Dentro deste contexto surgem as técnicas de *Business Intelligence* (BI). Elas assumem formas de ferramentas computacionais que

 Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

auxiliam a visualização, a análise e a interpretação das informações, com o intuito de que elas gerem conhecimento para as organizações. Baseado nisso, o objetivo deste artigo é abordar a ferramenta *FastCube* desenvolvida pela empresa *Fast Reports*. Tal ferramenta apresenta todas as características que possibilitam colocar em prática os processos de BI. Será construído um exemplo completo em Delphi utilizando o sistema gerenciador de banco de dados SQL Server, no qual serão abordados os principais recursos do *FastCube*, tais como: construção de modelo de dados multidimensional; visualização na forma de um cubo; filtragem; agrupamento; ordenação; formatação condicional; e definição e análise de indicadores de desempenho.

As duas próximas seções apresentam os conceitos básicos que envolvem BI, bem como os termos OLAP *(On-Line Analytical Processing*) e modelagem multidimensional. O entendimento desses conceitos é fundamental para o entendimento do *FastCube* e dos exemplos práticos abordados posteriormente.

Business Intelligence

As tecnologias de BI foram concebidas para prover vantagens informacionais às empresas em suas operações e decisões. Isso é obtido por meio de uma apresentação singular da informação para subsidiar estratégias de tomadas de decisões gerenciais/executivas. Em suma, um software de BI deve ser capaz de filtrar informações e transformá-las em conhecimento útil que possa caracterizar vantagens competitivas para as organizações. Além disso, o BI visa a criação de relatórios gerenciais para todos os níveis da organização e para todos os tipos de pessoas que tomam decisões dentro das empresas. Neste contexto, existem basicamente três classes de relatórios, que são: (i) *dashboard*; (ii) produção; e (iii) analítico. O *dashboard* é um relatório altamente sumarizado e formado por agregações, as quais podem apresentar informações gráficas do que está acontecendo na empresa. Em geral, esse tipo de relatório é utilizado em decisões estratégicas, como para decidir se é viável conceder um desconto para um determinado produto. A segunda classe de relatório, por sua vez, é em geral fornecida pelo próprio sistema e um exemplo é uma simples listagem dos produtos mais vendidos. Por fim, os relatórios analíticos possibilitam a interação com o usuário, como a adição dinâmica de colunas e valores personalizados. Desta forma, essa classe de relatório permite a utilização de ferramentas para mostrar cálculos complexos. Esse tipo de relatório fornece as informações mais valiosas para as organizações.

Em resumo, BI preocupa-se em transformar os dados em informação que as empresas possam utilizar em seus negócios, objetivando melhorar e facilitar seus processos gerenciais. Isso é obtido por meio da integração de diversas fontes de dados que podem existir em uma empresa, ou seja, bases de dados, arquivos de texto, planilhas eletrônicas, *log* de servidores, dentre outros. Essas fontes de dados podem ser agrupadas, e por meio desse agrupamento o gestor pode tirar conclusões sobre como a empresa encontrava-se em um determinado período, como encontra-se atualmente e também realizar previsões de como os negócios estarão no futuro. Por meio de BI, os gestores terão a capacidade de responder a uma série de questões relativas a determinadas áreas da empresa.

No contexto de BI existem quatro tecnologias fundamentais para sua operacionalização:
(i) banco de dados,
(ii) pacote de ferramentas de mineração de dados e estatísticas,
(iii) OLAP e
(iv) visualização. A primeira tecnologia são os bancos de dados, os quais necessitam estar preparados para a realização de análises.

O segundo item representa uma grande variedade de algoritmos de mineração de dados (e.g., árvores de decisão ou redes neurais), voltados a encontrar padrões. A terceira tecnologia é o OLAP, que proporciona uma visão multidimensional dos dados; e por fim, a visualização consiste de interfaces gráficas que possibilitam a interação dinâmica com os dados. No presente artigo serão abordados com detalhes o terceiro e o quarto item utilizando o *FastCube*. Com relação ao banco de dados, o mesmo será disponibilizado com a estrutura já definida para o presente estudo, enquanto que o processo de mineração de dados não será abordado. A próxima seção tem o objetivo de apresentar mais detalhes sobre os conceitos e processos de BI.

Data warehouse e bases de dados transacionais

Para que os dados sejam transformados em informações para posteriormente serem analisadas, é necessário que a base de dados esteja em um esquema específico. Em geral, isso é obtido por meio da construção de um *data warehouse* (d/w), que é um banco de dados destinado somente para armazenar dados, no qual os valores são estáveis e consistentes. Os d/w são diferentes das bases de dados transacionais encontradas nas empresas, ou seja, enquanto que uma base de dados transacional armazena as atividades diárias da empresa (incluir, alterar e excluir), um d/w guarda informações históricas geralmente de longa data. Algumas características de um d/w são: ser voltado ao planejamento, possuir alto nível de agregação, apresentar dados integrados, armazenar as atividades durante o tempo, ser estável e apresentar rapidez para agregações. Dentro deste contexto também existe o termo *Data Mart* (d/m), que é considerado um subconjunto do d/w. Por exemplo, uma empresa pode possuir um d/w com informações sobre as vendas de produtos e um d/m sobre as vendas somente de um departamento específico. Desta forma, um d/m é definido como uma subdivisão ou fatia do d/w.

Para que as técnicas de BI possam ser aplicadas em um d/w, são necessárias ferramentas OLAP. Esse termo é utilizado para representar ferramentas que permitem a realização de estudos analíticos nos dados. Uma ferramenta OLAP, tal como o *FastCube*, permite que os dados sejam analisados sob diversas perspectivas e apresenta as informações utilizando diversas dimensões. É importante salientar que antes de utilizar essas ferramentas, é necessário que seja realizada uma etapa de preparação, chamado de modelagem multidimensional.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Em resumo, é preciso transformar a base de dados transacional em um d/w que contenha somente os dados que são importantes para as análises. Em uma modelagem multidimensional existem três conceitos fundamentais, que são:
(i) medida;
(ii) dimensão; e
(iii) cubo.

As medidas são valores numéricos, como a quantidade de vendas, a quantidade de acessos, a média de investimentos ou a soma de salários. As medidas são dependentes do contexto e a **Tabela 1** ilustra duas medidas: as unidades vendidas e o valor total vendido. Com relação às unidades vendidas, neste exemplo elas estão representadas por caixas vendidas de cada produto, ou seja, uma unidade equivale a uma caixa. São os valores das medidas que são avaliados e analisados pelos gestores com a ajuda das ferramentas OLAP.

Por outro lado, uma dimensão é uma maneira de melhorar a exibição das medidas. Por exemplo, nessa tabela existem três dimensões: o estado, o produto e o mês. Essas três dimensões estão sendo utilizadas em conjunto, ou seja, pode-se concluir que foram vendidas 13 caixas de camarão ao valor de R$ 230,00 no estado de São Paulo e no mês de janeiro. Quanto mais dimensões forem adicionadas ao modelo, mais detalhadas as informações serão para a tomada de decisão; as dimensões afetam os valores das medidas. Pode-se observar que cada uma das três dimensões possui diversos valores, ou seja, a dimensão mês é composta pelos valores: janeiro, fevereiro, março e abril. É importante salientar que uma dimensão pode apresentar vários valores de atributos. O mesmo ocorre para a dimensão estado, que possui os valores PR, SP e SC e para a dimensão produto, que contém os seguintes valores: legumes, camarão e sardinha.

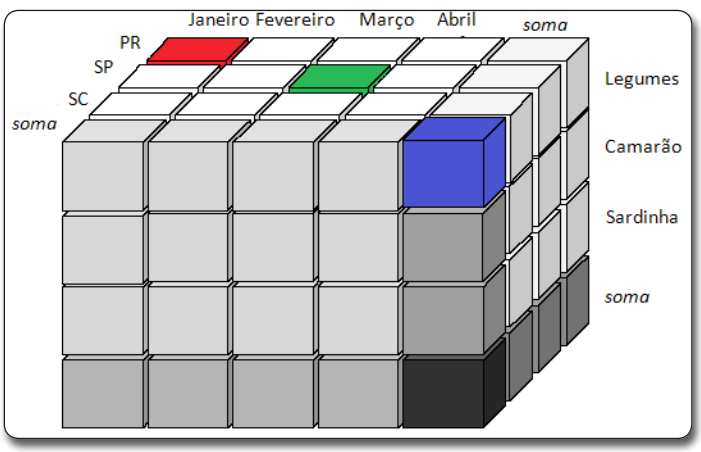


Figura 1. Visualização multidimensional de dados – um cubo

Por fim, o cubo é a forma multidimensional de apresentação dos dados nos quais as análises são realizadas. O ilustrado por meio da **Figura 1** apresenta a mesma informação da **Tabela 1**.

Neste cubo é possível observar as três dimensões; e cada célula representa um valor. Para exemplificação, será considerada somente a medida relacionada às unidades vendidas, e, por exemplo, a célula vermelha representa a quantidade de caixas de legumes vendidas no estado do Paraná no mês de janeiro, correspondendo a 10 unidades pela **Tabela 1**. De forma similar, a célula verde representa a quantidade de caixas de camarão vendidas no estado de São Paulo no mês de março, correspondendo a 14 unidades. Por fim, a célula azul representa um somatório, que indica o total de caixas de legumes vendidas em todos os estados e em todos os meses, correspondendo a 97 unidades. Na figura, as células brancas correspondem a valores individuais, enquanto que as células na cor cinza dizem respeito à agregação de valores, tais como somas, médias ou contagens.

O conceito de cubo em BI é um dos mais importantes, pois é por meio dele que todas as análises, proporcionadas pelas ferramentas OLAP, são realizadas. As próximas seções mostram como essas análises podem ser feitas por meio do SQL Server, do Delphi e do *FastCube*.

Entendendo a base de dados

A **Figura 2** apresenta o diagrama entidade-relacionamento. Pode-se observar que existem quatro tabelas que representam as dimensões (cliente, loja, classe de produtos e tempo) e uma tabela no centro do diagrama que contém as medidas. De acordo com o esquema multidimensional, a tabela que possui as medidas é chamada de *tabela fato*, à medida que é nela que estão armazenados os valores numéricos que serão posteriormente agregados e analisados. Este formato de diagrama está relacionado ao modelo

Estado	Produto	Janeiro		Fevereiro		Março		Abril	
		Unidade	Valor	Unidade	Valor	Unidade	Valor	Unidade	Valor
PR	Legumes	10	50,00	15	85,00	02	28,00	03	32,00
	Camarão	05	95,00	02	35,00	09	128,00	10	135,00
	Sardinha	07	55,00	09	85,00	10	95,00	05	45,00
SP	Legumes	12	120,00	07	68,00	05	38,00	12	120,00
	Camarão	13	230,00	10	190,00	14	240,00	15	255,00
	Sardinha	02	7,00	20	107,00	10	70,00	04	14,00
SC	Legumes	08	30,00	11	66,00	05	70,00	07	28,00
	Camarão	10	190,00	10	190,00	15	260,00	10	190,00
	Sardinha	04	14,00	15	85,00	09	63,00	05	17,00

Tabela 1. Relatório com medidas e dimensões

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

estrela, no qual a *tabela fato* encontra-se no centro e todas as dimensões estão a ela ligadas.

Pode-se observar também que o diagrama segue um formato um pouco diferente de uma modelagem tradicional; essa última segue as formas normais. Isso acontece porque, como escrito anteriormente, uma base de dados transacional necessita ser convertida para um d/w, e para isso, um esquema multidimensional deve ser aplicado com o intuito de obter um melhor desempenho das ferramentas OLAP. Na sequência algumas dessas diferenças serão discutidas com maiores detalhes.

O esquema da figura visa apresentar os dados de vendas de produtos em diferentes mercados, sendo que cada registro da *tabela fato* representa uma transação de venda. Essa tabela é composta pela classe do produto, a data da venda, o cliente e a loja na qual a transação foi realizada, apresentando um total de 86.837 registros. As medidas desta tabela são os campos *unidades vendidas* e *valor vendido*. As análises e os agrupamentos serão feitos com base nesses dois campos numéricos. Como visto nas seções anteriores, para proporcionar uma melhor interpretação e visualização das medidas, são necessárias dimensões; as quais afetam a maneira como as medidas são visualizadas e agrupadas. Na sequência, cada uma das quatro dimensões é explanada.

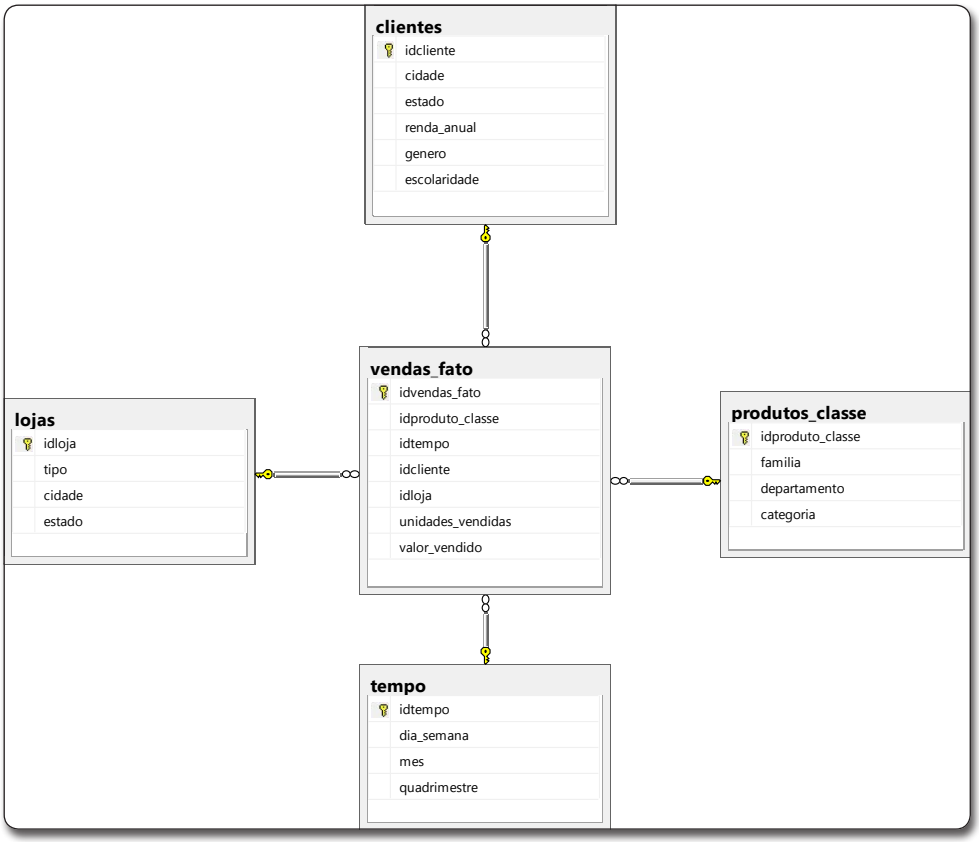


Figura 2. D/W utilizado no exemplo prático

A dimensão cliente é composta por 10.281 registros, na qual cada registro corresponde a um cliente distinto que realizou uma ou mais compras. Os atributos desta dimensão são: cidade, estado, renda anual, gênero (M ou F) e escolaridade. A renda anual é composta por oito intervalos: R$ 10.000 a R$ 30.000, R$ 30.000 a R$ 50.000, R$ 50.000 a R$ 70.000, R$ 70.000 a R$ 90.000, R$ 90.000 a R$ 110.000, R$ 110.000 a R$ 130.000, R$ 130.000 a R$ 150.000 e + do que R$ 150.000. Por outro lado, o domínio do atributo escolaridade é formado pelas seguintes categorias: ensino médio incompleto, ensino médio completo, ensino superior incompleto, ensino superior completo e pós-graduação completo. Com relação aos atributos cidade e estado, ambos são do tipo *varchar* para reduzir a complexidade do diagrama e também o processamento para executar as consultas. Em uma base de dados transacional que faça uso das formas normais, esses dois campos poderiam ser modelados como chaves estrangeiras. Semelhante à dimensão cliente, a dimensão loja contém 25 registros que representam as lojas nas quais as vendas foram efetuadas. Os atributos cidade e estado são semelhantes aos anteriores, enquanto que o atributo tipo tem o seu domínio formado pelos seguintes valores: supermercado, supermercado de luxo, supermercado gourmet, mercearia grande, mercearia pequena e sede da empresa.

A dimensão *produto classe* é composta por 110 registros e representa a família, departamento e categoria de um determinado produto. Para exemplificar pode-se considerar a família de produtos Bebida que possui um departamento chamado Bebidas alcoólicas, e que, por sua vez, possui uma categoria de produtos chamada Cerveja e Vinho. Em outras palavras, pode-se dizer que a cerveja está dentro da categoria de bebidas alcoólicas, enquanto que as bebidas alcoólicas estão dentro da família bebidas. Conforme dito anteriormente, na modelagem multidimensional é possível que esses três campos sejam textuais, diferentemente de uma modelagem transacional, na qual essa mesma estrutura poderia ser construída por meio de uma tabela com auto relacionamento. É importante salientar que o objetivo principal de um d/w é viabilizar a análise de grandes volumes de dados, e a utilização de uma estrutura puramente transacional poderia afetar o desempenho da aplicação e poderia tirar sua propriedade de fornecer respostas rápidas.

Por fim, a dimensão tempo indica o dia da semana, o mês e o quadrimestre no qual cada venda foi efetuada. Em uma base de dados transacional esses dados seriam extraídos de um único campo do tipo data. Essa pormenorização de uma data é um processo bastante comum em modelagem multidimensional para permitir a realização de agrupamentos dos dados utilizando diversas perspectivas. Por exemplo, uma empresa de telecomunicações

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

pode estar interessada em informações mais detalhadas sobre as ligações telefônicas, e para isso, dados como horas, minutos e segundos podem ser muito importantes neste contexto.

A **Listagem 1** apresenta o comando SQL para recuperar todos os dados das cinco tabelas presentes no modelo entidade-relacionamento da **Figura 2**. Este comando realiza a junção dos dados de todas as tabelas e será utilizado posteriormente para a construção da aplicação no Delphi.

Listagem 1. SQL para a obtenção dos dados do d/w

```
01 select
02  cln.cidade as cliente_cidade,
03  cln.estado as cliente_estado,
04  cln.renda_anual as cliente_renda,
05  cln.genero as cliente_genero,
06  cln.escolaridade as cliente_escolaridade,
07  ljs.tipo as loja_tipo,
08  ljs.cidade as loja_cidade,
09  ljs.estado as loja_estado,
10 prc.familia as produto_familia,
11  prc.departamento as produto_departamento,
12  prc.categoria as produto_categoria,
13  tmp.dia_semana,
14  tmp.mes,
15  tmp.quadrimestre,
16  vnf.unidades_vendidas,
17  vnf.valor_vendido
18 from vendas_fato vnf
19 inner join clientes cln on vnf.idcliente = cln.idcliente
20 inner join lojas ljs on vnf.idloja = ljs.idloja
21 inner join produtos_classe prc on vnf.idproduto_classe = prc.idproduto_classe
22 inner join tempo tmp on vnf.idtempo = tmp.idtempo
```

Configurando o acesso aos dados no Delphi

Após ter compreendido a estrutura da base de dados, a próxima etapa é: (i) a criação de uma aplicação do tipo VCL no Delphi (*File>New>VCL Forms Application*); (ii) a configuração dos componentes de acesso à base de dados SQL Server em um DataModule (*File>New>Other>Delphi Files>DataModule*). A **Figura 3** apresenta os componentes que devem ser adicionados no DataModule (dmAcessoDados) para tal configuração, que são: TADOConnection (conexao), TADOQuery (query), TfcxDBDataSet (dataset) e TfcxDataSource (datasource). O componente ADOConnection é responsável por realizar a conexão com a base de dados; duplo clique sobre ele abre o editor de associação de dados e configuração do acesso ao SQL Server via assistente. O componente *query* deve ter sua propriedade *Connection* ligada ao componente de conexão, bem como o script SQL da **Listagem 1** deve ser inserido na propriedade SQL.

O componente TfcxDBDataSet (dataset) do conjunto de componentes do *FastCube* é responsável por carregar os dados de um descendente do TDataSet, ou seja, do ADOQuery neste exemplo. Para isso, a sua propriedade DataSet deve apontar para o componente *query*. Por fim, o componente TfcxDataSource (datasource) deve ter sua propriedade DataSet ligada no componente dataset; ela indica a principal fonte de dados para a posterior geração do cubo. Este é o componente que contém as descrições completas das estruturas de dados para o cubo, incluindo as fontes de dados, arquivos para essas fontes, relações entre fontes de dados e regras de conversão.

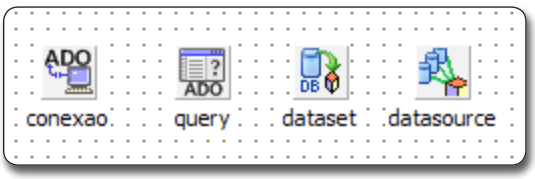


Figura 3. Acesso aos dados no Delphi

Criando a interface gráfica no Delphi

Concluída a configuração do acesso ao banco de dados, o próximo passo é construir a interface gráfica na qual o cubo será apresentado. Para isso, a aplicação conterá somente um formulário com um PageControl com duas guias. A primeira guia é a responsável por apresentar o cubo; e a essa guia devem ser adicionados os seguintes componentes: um TfcxSliceGridToolBar no topo da janela e um TfcxSliceGrid no restante da janela. O primeiro componente é uma barra de ferramentas pronta com operações que podem ser aplicadas ao cubo e o segundo componente é responsável pela exibição das medidas e dimensões.

Nota

São necessários dois passos para a completa instalação do FastCube. O primeiro é executar o instalador disponibilizado no site oficial da ferramenta (ver seção **Links**), para que a paleta de componentes básicos seja instalada no IDE. O segundo passo é utilizar o Recompile Wizard (recompile.exe) que está disponível na pasta de instalação para que as seguintes funcionalidades também sejam disponibilizadas na paleta de componentes: exportação dos dados para diversos formatos, gráficos e integração com o FastReport. Na seção **Links** encontra-se um artigo completo sobre o FastCube, sendo abordados os processos de instalação.

A segunda guia, por sua vez, é a responsável por apresentar os gráficos correspondentes às análises feitas na primeira guia, e os seguintes componentes devem ser adicionados: um TfcxChartToolBar e um TfcxChart. Semelhante aos componentes anteriores, o primeiro é uma barra de ferramentas pronta com operações pré-definidas para serem executadas sobre os gráficos e o segundo componente apresenta os gráficos. Maiores detalhes sobre esses componentes serão vistos na sequência.

A **Figura 4** apresenta a interface visual com as duas guias. Além desses componentes visuais, é necessário adicionar no formulário dois outros componentes não visuais, que são: um TfcxCube e um TfcxSlice. O primeiro componente é responsável por conectar-se à fonte de dados e buscar os dados e o TfcxSlice tem a função de particionar esses dados e facilitar a interação do usuário com as dimensões e medidas.

O componente *slice* deve ter sua propriedade Cube configurada para o componente *cubo*. Este último deve ter a sua propriedade DataSource ajustada para o TfcxDataSource. Além disso, o *cubo* deve ter sua propriedade CubeSource ajustada para fccs_DataSource. Isto indica que os dados são provenientes de uma conexão com uma base de dados.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

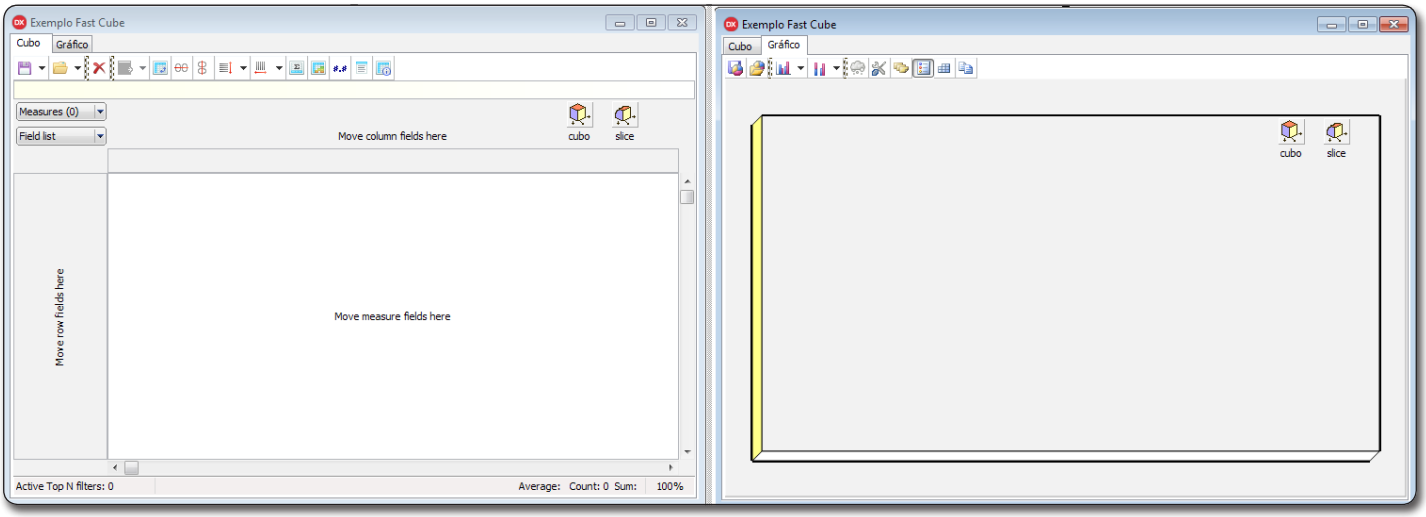


Figura 4. Interface gráfica do FastCube

Além dessa opção, o *FastCube* permite que o *cubo* seja gerado por meio de um arquivo próprio (fccs_CubeFile) ou por meio de *stream* de dados (fccs_CubeStream). Adicionalmente, tanto o componente fcxSlideGrid quanto o fcxChart devem ter sua propriedade *slice* configurada para o fcxSlice, enquanto que o fcxSliceGridToolBar e o fcxChartToolBar devem ser ligados ao fcxSliceGrid e fcxChart, respectivamente. O último passo é ativar o *cubo* no evento OnCreate do formulário por meio do seguinte código:

```
cubo.Active := True;.
```

Como visto, há várias ligações entre os componentes para que tanto a configuração de acesso aos dados quanto o formulário estejam completos. Para exemplificar as ligações entre eles, a **Figura 5** apresenta uma hierarquia dos componentes com as ligações que foram feitas entre eles.

Pode-se observar que a linha pontilhada na figura mostra a divisão entre os componentes de acesso a dados e os componentes relacionados à interface gráfica. É importante ressaltar que o componente TfcxDBDataSet conecta-se a qualquer classe descente de TDataSet, ou seja, ele pode ser utilizado com vários outros mecanismos de acesso a dados, tais como: FireDAC ou dbExpress.

Entendendo o esquema para apresentação e análise de dados

Após todas as configurações anteriores terem sido feitas o projeto já pode ser executado. A **Figura 6** apresenta um exemplo de um cubo com duas dimensões (classe do produto e tempo) e três medidas no centro da janela; pode-se também visualizar algumas totalizações. O componente fcxSliceGrid é o responsável pela apresentação dos dados; todo o processo para construir as diferentes visualizações dos dados depende do entendimento deste componente. As principais áreas do fcxSliceGrid estão numeradas na **Figura 6** e são explicadas a seguir, bem como as indicações para obter o mesmo resultado apresentado na figura em questão, já que por padrão o esquema/grade é inicializado em branco.

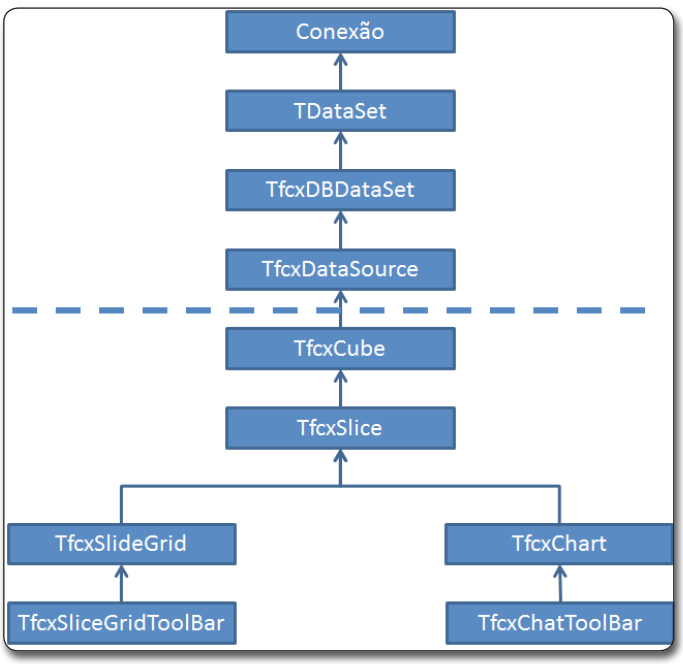


Figura 5. Ligação entre os componentes.

Na figura temos destacados alguns pontos:

1: Lista de campos (field list): todos os campos da consulta feita anteriormente na base de dados são listados nesta caixa de seleção. Por meio desta opção são escolhidos quais campos farão parte da análise – medidas, dimensões e hierarquias das dimensões.

Nota

É importante distinguir as diferenças entre dimensão e atributos da dimensão. Por exemplo, no diagrama entidade relacionamento da **Figura 2** existem quatro dimensões, sendo que cada uma delas possui um ou mais atributos. De acordo com as convenções em OLAP, tais atributos podem ser agrupados na forma de hierarquias. Por exemplo, uma cidade está dentro de um estado (dimensão cliente e loja), um dia da semana está dentro de um mês (dimensão tempo). Esse segundo exemplo pode ser visto na **Figura 6**.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

2: Região dos filtros (filter region): os atributos inclusos nesta região podem ser utilizados para realizar a filtragem dos dados apresentados na área central.
3: Cabeçalho das dimensões verticais (vertical dimension headers): as dimensões inclusas nesta região formam o cabeçalho vertical do cubo. Na **Figura 6** essas dimensões são representadas pelo atributo departamento do produto da dimensão classe do produto (alimentos enlatados e alimentos para café da manhã), e pelos atributos mês e dia da semana da dimensão tempo. Para incluir essas dimensões nesta área, basta selecionar os respectivos campos da lista de campos e movê-los para a área indicada como "move row fields here" do fcxSlideGrid.
4: Cabeçalho vertical da grade (vertical header of the grid): nesta área o agrupamento entre as dimensões é realizado, conforme pode ser observado na **Figura 6**. Por exemplo, foram vendidas 46 unidades no departamento de alimentos para café da manhã nos domingos do mês de abril. Similarmente, o valor total vendido de alimentos enlatados em janeiro foi de R$ 1.274,49, conforme pode ser observado no quadro em vermelho desta figura. É importante salientar que a ordem dos campos adicionados na região 3 faz diferença no resultado final apresentado, ou seja, se o atributo mês for arrastado para frente do atributo departamento, os agrupamentos serão primeiramente realizados por este campo.
5: Cabeçalho das dimensões horizontais (horizontal dimension headers): as dimensões inclusas nesta região formam o cabeçalho horizontal do cubo. Na **Figura 6** isso é representado pelas medidas (measures). Para obter o mesmo efeito desta figura, deve-se arrastar da lista de campos as medidas existentes na base de dados, que são: *unidades vendidas* e *valor vendido* para a área indicada como "move measures fields here" do fcxSliceGrid. Além desses dados, o próprio *FastCube* adiciona também uma medida para a contagem dos registros, indicado como "system counter". Após arrastar os campos para a área central é necessário arrastar a caixa de seleção - Measures (3) – para este cabeçalho horizontal, para que assim os valores sejam automaticamente totalizados e agrupados.
6: Região de dados (data region): como visto anteriormente, essa é a região na qual os dados agrupados e totalizados são apresentados.

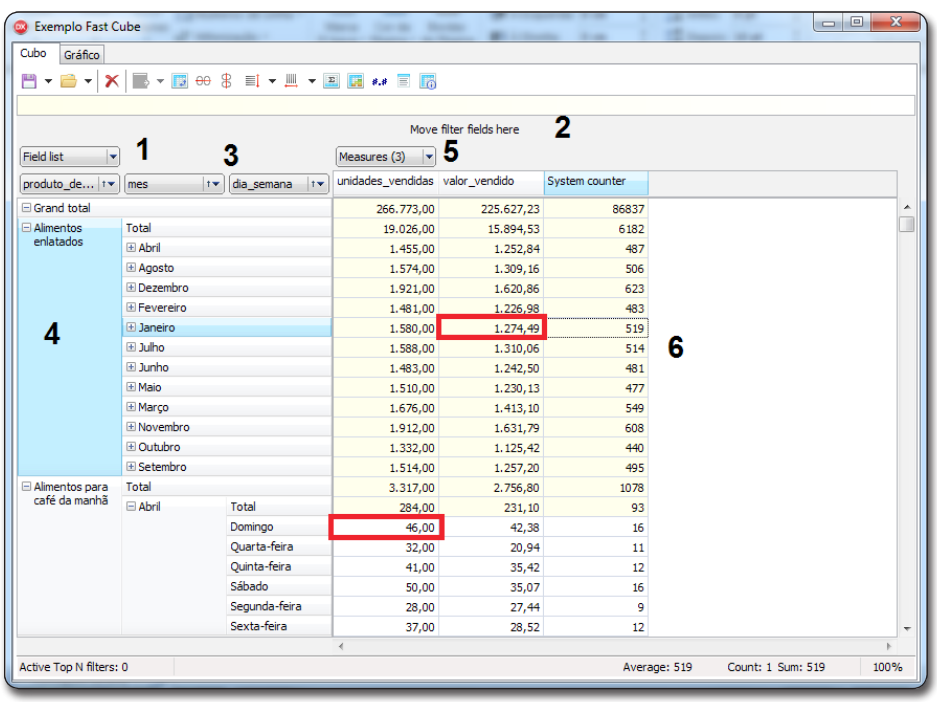


Figura 6. Composição do fcxSliceGrid

Nota

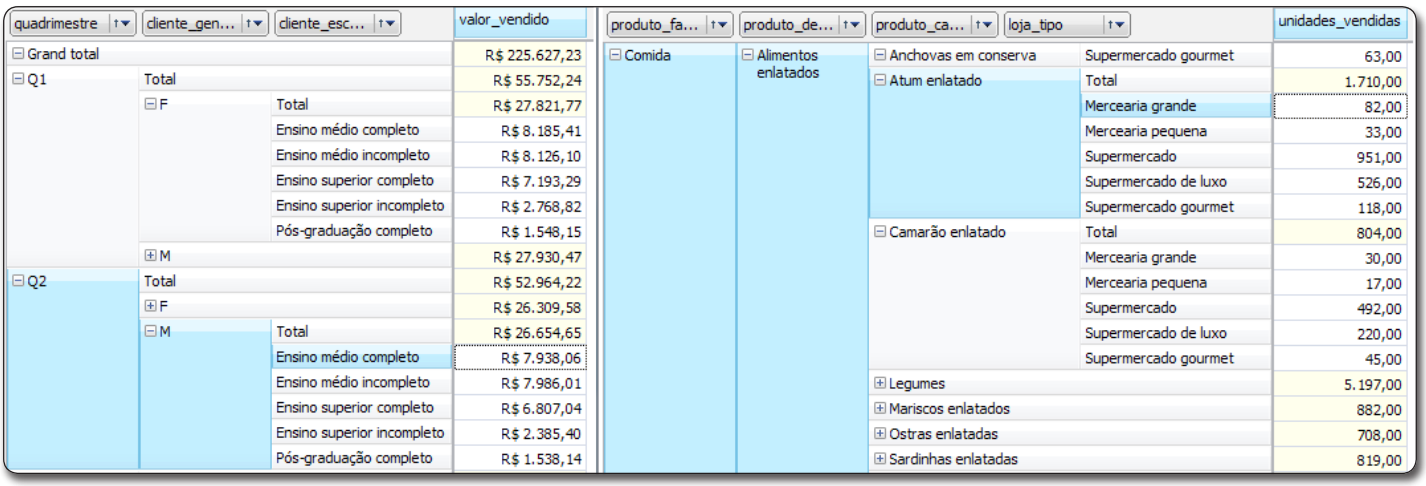
O layout da grade pode ser alterado por meio de um conceito de OLAP chamado de rotação. Esse mecanismo consiste em rotacionar um vetor multidimensional, de modo que as dimensões verticais (item 4 da **Figura 6**) assumam o lugar das dimensões horizontais (item 5). O componente fcxSliceGridToolBar possui o botão transpose grid, o qual realiza essa tarefa de forma automática. Em suma, a rotação permite que os mesmos dados sejam analisados sob uma perspectiva diferente.

A **Figura 7** apresenta outros dois exemplos de cubos que podem ser gerados seguindo os mesmos procedimentos descritos anteriormente. O exemplo da esquerda visa analisar a medida *valor vendido*, tendo como atributos o quadrimestre da dimensão *tempo* e, *gênero* e *escolaridade* da dimensão *cliente*. Por meio deste tipo de análise pode-se responder questões do tipo: qual a influência do gênero e da escolaridade face ao montante total de produtos vendidos? Há algum tipo de relação entre o quadrimestre em que a venda foi efetuada e o gênero dos clientes? Há alguma ligação entre a escolaridade do cliente e o valor vendido? Por meio da análise dos valores, questões como essa podem ser respondidas, bem como padrões nos dados podem ser descobertos. Por exemplo, com base nos valores totais apresentados nos dois quadrimestres (Q1 e Q2), pode-se chegar à conclusão de que quanto maior o nível de escolaridade, menos os clientes gastam; visto que em ambos os casos as pessoas com escolaridade de ensino superior e pós-graduação apresentam menores valores de compra.

Por outro lado, o cubo do lado direito da **Figura 7** apresenta a medida *unidades vendidas*, tendo como atributos de análise a família, o departamento e a categoria da dimensão *classe do produto* e também o tipo da dimensão *loja*. Baseado neste cubo, diversas análises podem ser feitas, por exemplo: em qual tipo de estabelecimento é recomendável fornecer mais atum enlatado? É viável vender camarão enlatado em pequenas mercearias?

Como visto nesses exemplos, diversas análises e agrupamentos podem ser feitos utilizando as dimensões e as medidas;

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia



quadrimestre	cliente_gen...	cliente_esc...	valor_vendido
Grand total			R$ 225.627,23
Q1	Total		R$ 55.752,24
	F	Total	R$ 27.821,77
		Ensino médio completo	R$ 8.185,41
		Ensino médio incompleto	R$ 8.126,10
		Ensino superior completo	R$ 7.193,29
		Ensino superior incompleto	R$ 2.768,82
		Pós-graduação completo	R$ 1.548,15
	M		R$ 27.930,47
Q2	Total		R$ 52.964,22
	F		R$ 26.309,58
	M	Total	R$ 26.654,65
		Ensino médio completo	R$ 7.938,06
		Ensino médio incompleto	R$ 7.986,01
		Ensino superior completo	R$ 6.807,04
		Ensino superior incompleto	R$ 2.385,40
		Pós-graduação completo	R$ 1.538,14

produto_fa...	produto_de...	produto_ca...	loja_tipo	unidades_vendidas
Comida	Alimentos enlatados	Anchovas em conserva	Supermercado gourmet	63,00
		Atum enlatado	Total	1.710,00
			Mercearia grande	82,00
			Mercearia pequena	33,00
			Supermercado	951,00
			Supermercado de luxo	526,00
			Supermercado gourmet	118,00
		Camarão enlatado	Total	804,00
			Mercearia grande	30,00
			Mercearia pequena	17,00
			Supermercado	492,00
			Supermercado de luxo	220,00
			Supermercado gourmet	45,00
		Legumes		5.197,00
		Mariscos enlatados		882,00
		Ostras enlatadas		708,00
		Sardinhas enlatadas		819,00

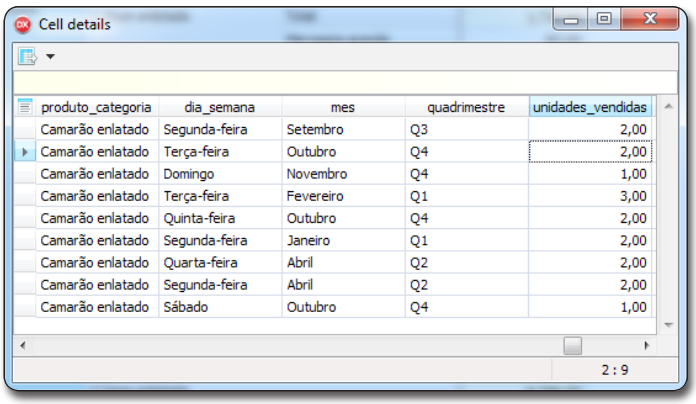
Figura 7. Exemplos de cubos no FastCube

isto é feito por meio da combinação das mesmas nos cabeçalhos das dimensões horizontais ou verticais (itens 3 e 5 da **Figura 6**). Apesar do d/w utilizado neste exemplo possuir apenas quatro tabelas para representar as dimensões, um grande número de combinações pode ser feito para alimentar diferentes maneiras de interpretar os dados, ajustando-se a perspectiva que o gestor deseja visualizar as informações.

Nota

No componente fcxSliceGridToolBar estão disponíveis opções para gravar e carregar cubos previamente construídos. Os arquivos são gravados com a extensão mdc. Eles gravam os dados, os filtros e as configurações de formatação. Essa opção é útil quando um cubo é criado e deseja-se analisar os dados em um momento futuro. Para limpar o componente fcxSliceGrid e iniciar um novo cubo, deve-se utilizar a opção clear formatting scheme. Além dessas opções, a barra de ferramentas apresenta outras funções úteis, tais como: exportar dados para outros formatos, ocultar atributos com valores nulos, ordenar os dados por linhas ou colunas e configurar a formatação das medidas.

Pode-se notar, nos cubos mostrados anteriormente, que há botões "+" e "-" para os atributos que possuem outros atributos relacionados, como o tipo da loja que está "dentro" do atributo categoria (conforme **Figura 7**). Neste contexto, existem dois conceitos importantes em OLAP: *drill down* e *drill up*. Em outras palavras, quando o usuário clica no botão "+" ao lado categoria Legumes, todos os dados dos tipos de loja relacionados a esta categoria serão expandidos e exibidos, ou seja, foi feita uma navegação "para baixo" na direção do dado mais detalhado. Por outro lado, quando o botão "-" ao lado de Atum enlatado for clicado, os dados relativos aos tipos de loja serão escondidos, ou seja, foi feita uma navegação "para cima" na direção do dado mais resumido. Além desses conceitos, o *FastCube* trabalha também com o *drill through*, que permite realizar uma inspeção detalhada dos dados que compõem uma medida. Por exemplo, dando duplo clique no valor 17 (Mercearia pequena>Camarão enlatado), é possível visualizar os dados mostrados pela **Figura 8.** Cada um dos nove registros apresentados compõe o valor total de 17 unidade vendidas.



produto_categoria	dia_semana	mes	quadrimestre	unidades_vendidas
Camarão enlatado	Segunda-feira	Setembro	Q3	2,00
Camarão enlatado	Terça-feira	Outubro	Q4	2,00
Camarão enlatado	Domingo	Novembro	Q4	1,00
Camarão enlatado	Terça-feira	Fevereiro	Q1	3,00
Camarão enlatado	Quinta-feira	Outubro	Q4	2,00
Camarão enlatado	Segunda-feira	Janeiro	Q1	2,00
Camarão enlatado	Quarta-feira	Abril	Q2	2,00
Camarão enlatado	Segunda-feira	Abril	Q2	2,00
Camarão enlatado	Sábado	Outubro	Q4	1,00

2 : 9

Figura 8. Drill through

Visualizando gráficos

Na segunda guia do formulário principal foi adicionado um componente fcxChart. Ele tem a função de apresentar gráficos de acordo com os dados presentes no componente fcxSliceGrid da primeira guia. Para que isso seja possível, o componente fcxChart deve ter sua propriedade Active ajustada para True. Desta forma, toda vez que houver alguma alteração nos dados o gráfico será automaticamente atualizado. A **Figura 9** apresenta os gráficos que correspondem aos cubos definidos na **Figura 7**.

Pode-se observar que os totais são gerados somente para os primeiros atributos que estão no cabeçalho das dimensões verticais, que são: o quadrimestre no gráfico da esquerda e a família de produtos no gráfico da direita. Acima do gráfico encontra-se o componente fcxChatToolBar, o qual é uma barra de ferramentas para manipulação do gráfico. Alguns exemplos de opções disponíveis são: gravar e carregar o gráfico, alterar o tipo e o estilo, assim como adicionar e retirar legendas. Outras configurações mais avançadas podem ser realizadas por meio da opção Chart Properties.

Filtrando, ordenando, totalizando e agrupando

O recurso de filtragem de dados no *FastCube* tem por objetivo excluir das análises valores de dimensões que não são interessan-

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

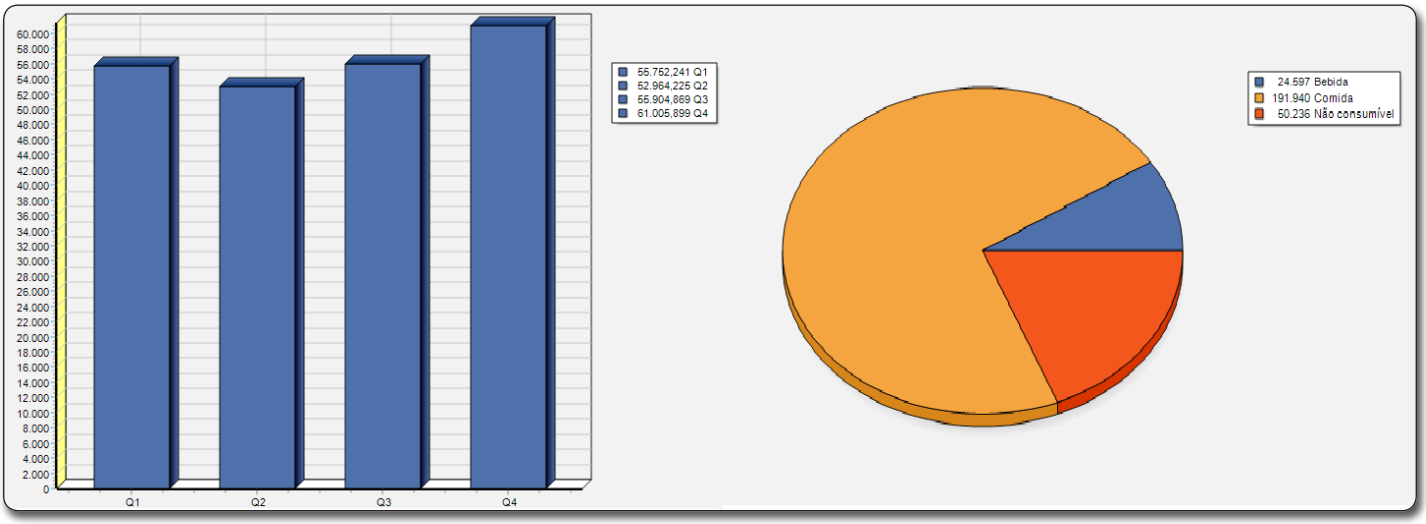


Figura 9. Apresentação de gráficos

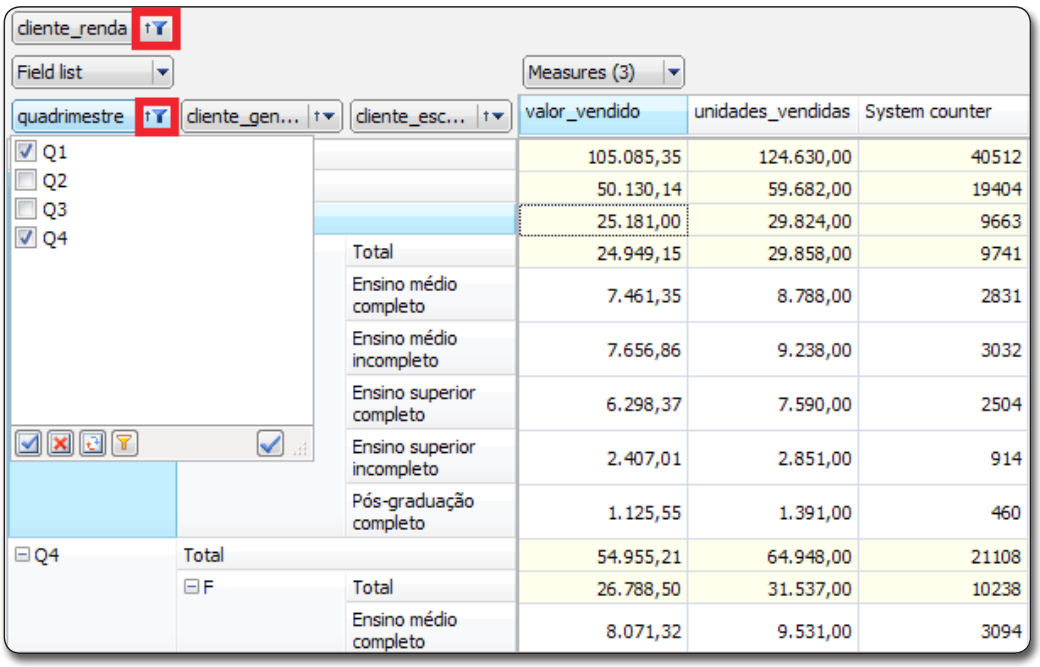


Figura 10. Aplicação de filtros

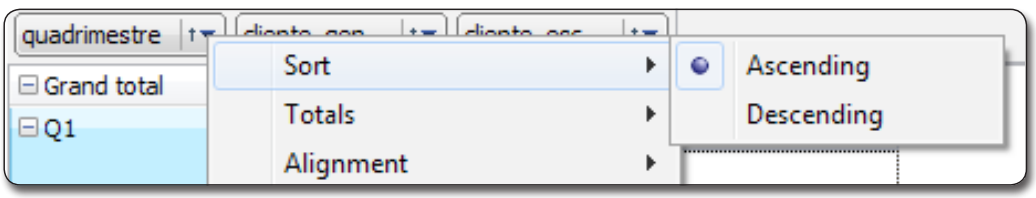


Figura 11. Ordenação dos dados

tes para o usuário. Para isso, um clique no atributo faz com que uma janela seja aberta para escolha dos valores que farão parte das totalizações das medidas. A **Figura 10** mostra que foram selecionados apenas os quadrimestres Q1 e Q4, e, por consequência, apenas os registros desses dois quadrimestres farão parte da análise. Os campos que apresentam filtros possuem um ícone correspondente, conforme retângulo em vermelho nesta figura. Pode-se notar que o atributo *cliente renda* está na parte superior do componente fcxSliceGrid. Isto indica que as medidas serão filtradas conforme este campo, porém, ele não faz parte da estrutura do cubo.

Por outro lado, a ordenação dos dados é obtida simplesmente clicando com o botão direito no atributo, conforme mostra a **Figura 11**.

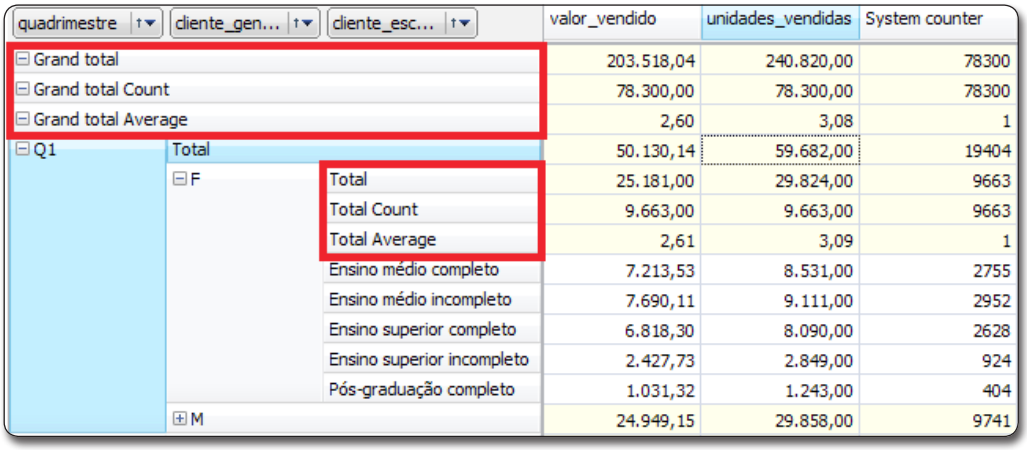
Como visto nos exemplos anteriores, além do componente fcxSliceGrid apresentar os valores relacionados a cada atributo de uma dimensão, são também mostrados valores totais relacionados a um grupo ou hierarquia. É exibido, por padrão, apenas uma totalização por cada grupo. Por exemplo, na **Figura 12** são apresentados apenas os campos Grand total na primeira linha e o Total ao lado do gênero F. O valor de 203.518,04 representa o somatório de todas as vendas e o valor 25.181,00 representa o somatório dos valores

Nota

É possível também realizar filtros personalizados e avançados que levam em consideração várias condições e elementos lógicos, clicando com o botão direito no atributo e selecionando a opção Custom Filter

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

de venda somente do sexo feminino no primeiro quadrimestre (Q1). Por outro lado, o valor 50.130,14 ao lado de Q1 indica o somatório das vendas no primeiro quadrimestre (Q1), independente do sexo. Essas totalizações correspondem aos conceitos teóricos apresentados no exemplo de cubo da **Figura 1** e **Tabela 1**.

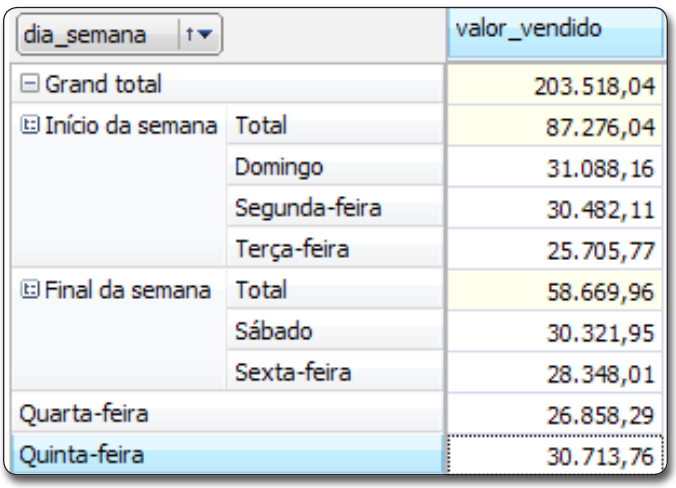


quadrimestre	cliente_gen...	cliente_esc...	valor_vendido	unidades_vendidas	System counter
Grand total			203.518,04	240.820,00	78300
Grand total Count			78.300,00	78.300,00	78300
Grand total Average			2,60	3,08	1
Q1	Total		50.130,14	59.682,00	19404
	F	Total	25.181,00	29.824,00	9663
		Total Count	9.663,00	9.663,00	9663
		Total Average	2,61	3,09	1
		Ensino médio completo	7.213,53	8.531,00	2755
		Ensino médio incompleto	7.690,11	9.111,00	2952
		Ensino superior completo	6.818,30	8.090,00	2628
		Ensino superior incompleto	2.427,73	2.849,00	924
		Pós-graduação completo	1.031,32	1.243,00	404
	M		24.949,15	29.858,00	9741

Figura 12. Totalizações no cubo

É possível observar nessa figura que além dos somatórios padrões, existem mais duas linhas destacadas em vermelho na parte superior, que são: Grand total Count e Grand total Average. O primeiro campo apresenta a contagem dos registros e o segundo apresenta as médias para cada uma das medidas. Esses campos podem ser adicionados conforme a necessidade da análise, e para incluí-los deve-se clicar com o botão direito na área dos cabeçalhos verticais (item 4 da **Figura 6**), selecionando a opção de menu Grand total. Nesta lista é possível escolher várias opções já pré-definidas, tais como mínimo, máximo, multiplicação de campos, desvio padrão, média, contagem de registros únicos, dentre outros. De forma similar é possível observar os campos Total Count e Total Average que dizem respeito às totalizações apenas para o atributo quadrimestre, e devem ser adicionados de forma similar clicando com o botão direito no atributo que está localizado no cabeçalho das dimensões verticais (item 3 da **Figura 6**).

Todos os agrupamentos e hierarquias apresentados anteriormente são baseados na lista de campos disponíveis na consulta em SQL feita anteriormente (item 1 da **Figura 6**). Porém, outra funcionalidade bastante útil do *FastCube* é o agrupamento dos dados em categorias que podem ser definidas em tempo de execução, conforme mostrado na **Figura 13**. Foram definidos dois grupos, um para representar o início da semana e outro para representar o final de semana, sendo que todas as totalizações são realizadas de forma similar aos atributos das dimensões existentes na consulta em SQL. É possível também que alguns registros não se enquadrem em nenhum grupo, como os dias da semana: quarta-feira e quinta-feira. Para criar grupos no *FastCube* deve-se clicar com o botão direito no cabeçalho vertical da grade (item 4 da **Figura 6**) e acessar a opção *Move to group>Create new*. Após um grupo ter sido criado, cada registro deve ser adicionado manualmente no grupo correspondente por meio desta mesma opção.



dia_semana		valor_vendido
Grand total		203.518,04
Início da semana	Total	87.276,04
	Domingo	31.088,16
	Segunda-feira	30.482,11
	Terça-feira	25.705,77
Final da semana	Total	58.669,96
	Sábado	30.321,95
	Sexta-feira	28.348,01
Quarta-feira		26.858,29
Quinta-feira		30.713,76

Figura 13. Criação de grupos manualmente

Formatando as medidas

Por padrão, as medidas são formatadas de acordo com o tipo de campo e precisão. Todavia, é possível alterar a apresentação dos dados por meio do editor de medidas (**Figura 14**). Para acessar o editor, deve-se clicar na medida em que se deseja aplicar um formato na região dos dados (item 6 da **Figura 6**) e depois clicar no botão correspondente na barra de ferramentas. O editor de medidas possui

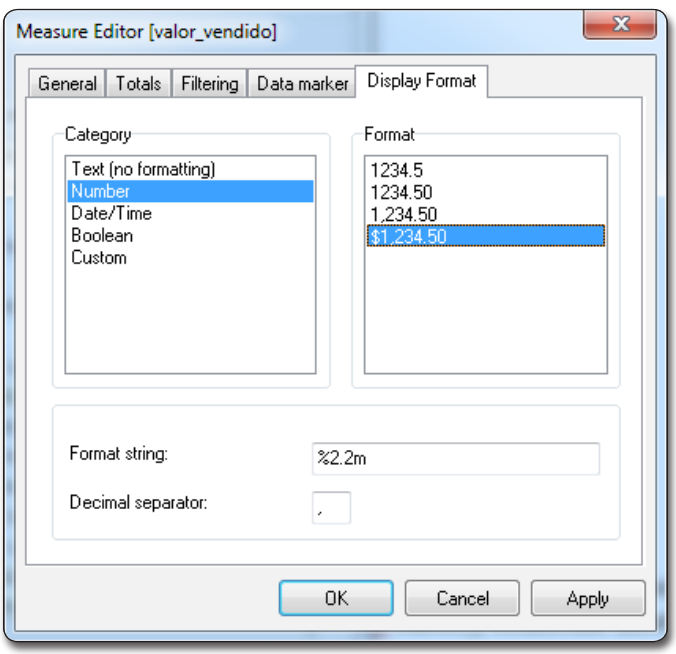


Figura 14. Aplicação de formatação nas medidas

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

várias opções, e uma delas é a configuração de formato (display format), que permite a configuração da categoria, formato, string de formatação e separadores decimais. No exemplo da **Figura 14** foi aplicado o formato de moeda para a medida *valor vendido*.

Na janela do editor de medidas também é possível destacar os dados por meio de expressões condicionais que podem ser configuradas na guia *Data marker*. Nesta guia é possível aplicar formatações dependendo dos valores das medidas. Por exemplo, na **Figura 15** foi definida uma condição que se o valor da célula *for maior ou igual a* 5.000, a célula terá seu estilo mudado para cor de fundo verde. É possível também observar que no lado direito desta figura pode-se definir se a formatação será aplicada somente para as células ou para os valores totais também.

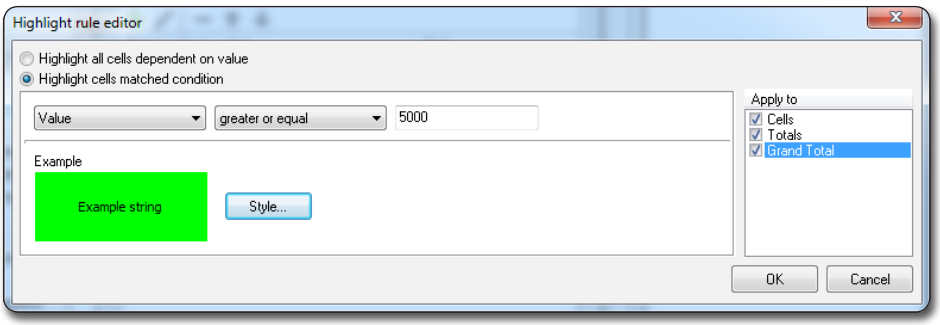


Figura 15. Expressões condicionais

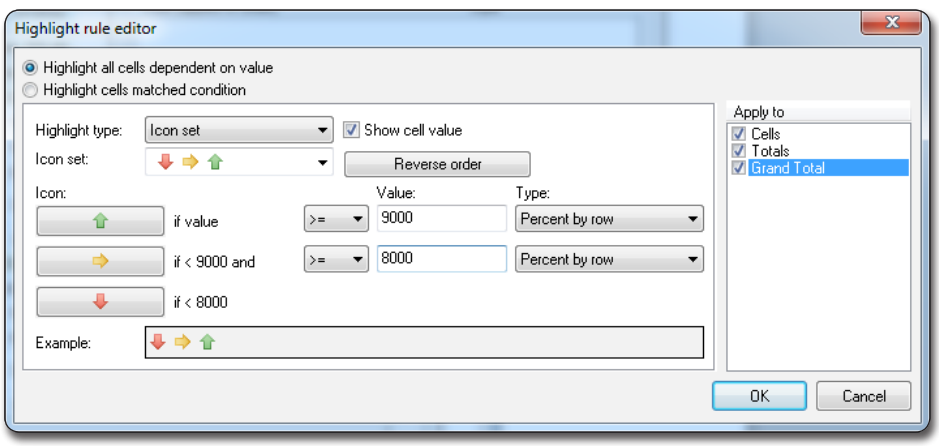


Figura 16. Configuração do realce contínuo das medidas

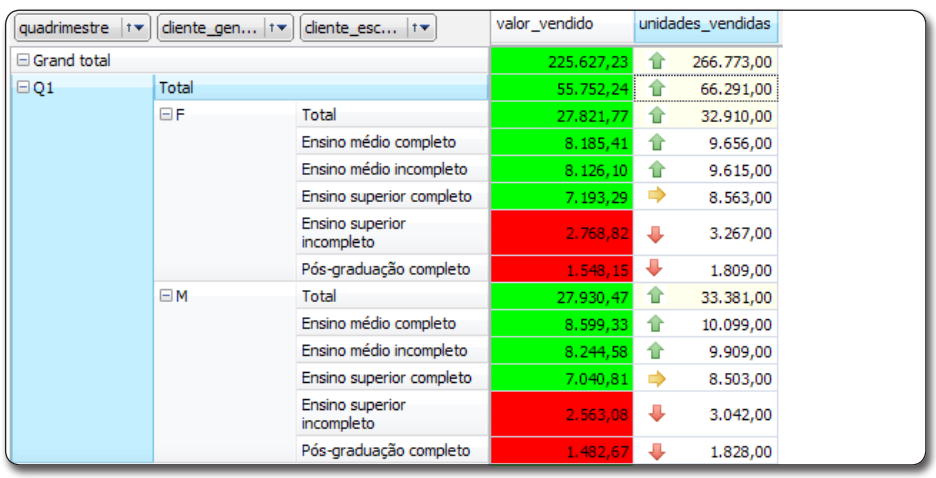


Figura 17. Visualização das formações condicionais e metas

Na figura foi escolhida a opção *Highlight cells matched condition*, que diz respeito à formatação condicional de célula de acordo com as regras definidas. Além deste tipo de formatação, o *FastCube* também permite o realce contínuo de células. Isto é obtido pela opção *Highlight all cells dependent on value*. Essa opção apresenta maneiras mais avançadas para realce dos dados. A **Figura 16** apresenta um exemplo de configuração que utiliza ícones para indicar *se a medida unidades vendidas está* acima, abaixo ou dentro de uma determinada meta especificada. Neste exemplo, caso a quantidade vendida seja maior que 9.000, será exibido o ícone verde com a seta para cima. Caso as vendas estejam entre 8.000 e 8.999 será exibida a seta horizontal amarela, indicando que as vendas estão estagnadas ou que não houve progresso. Por fim, valores menores que 8.000 indicam que as vendas diminuíram, sendo representadas pela seta vermelha.

Nota

O FastCube também disponibiliza mais duas formas para indicar a progressão das medidas, que são: escala de cores (gradiente) e barras. O gradiente tem a função de pintar as células com cores mais fracas ou fortes dependendo dos valores de uma medida, ou seja, valores menores terão um gradiente mais fraco, enquanto que valores maiores terão gradiente mais acentuado. Com relação às barras, elas são apresentadas como se fossem barras de progresso semelhantes ao componente ProgressBar, que indicam a progressão dos dados de acordo com os valores mínimos e máximos.

A **Figura 17** apresenta um cubo com os dois tipos de realce, que são: a formatação condicional simples para a medida *valor vendido* e o realce contínuo apresentado pelas setas. Deve-se notar que foi adicionada mais uma regra para a primeira medida, que consiste em pintar as células de vermelho quando os valores são menores do que 5.000.

É possível observar que as formatações são úteis para que os gestores consigam visualizar os dados de forma intuitiva e rápida. O exemplo da formatação contínua com as setas diz respeito ao conceito de KPI (*Key Performance Indicator*), o qual está

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

diretamente relacionado à tomada de decisão, à medida que por meio deste conceito podem ser definidas metas e acompanhar se elas estão sendo cumpridas, de forma online. Nesta mesma figura, pode-se notar que as vendas no primeiro quadrimestre para pessoas com ensino superior e pós-graduação não atingiram a meta estabelecida (seta vermelha). Isso pode ser um indicativo de que estratégias de mercado podem ser colocadas em prática para este público alvo ou então chegar-se à conclusão de que pessoas com este grau de escolaridade podem de fato não estarem interessadas nos produtos oferecidos pelas empresas analisadas.

A construção deste modelo de dados multidimensional – ilustrado na forma de um cubo – somente é viável graças à utilização de uma ferramenta OLAP para gerenciar medidas e dimensões de forma muito eficiente e amigável. OLAP é uma das tecnologias fundamentais para o universo do BI, como apresentado anteriormente. Ela viabiliza efetivamente os agrupamentos e os processamentos dos dados. Por outro lado, a visualização dos dados neste formato específico diz respeito à quarta tecnologia do BI, que são as ferramentas para visualização e análise dos dados.

O *FastCube* apresenta um conjunto de componentes bastante útil e consistente para realizar análises de dados, sem haver a necessidade de escrever muitas linhas de código para construir uma aplicação completa de BI. A grande vantagem do *FastCube* é o fato de ser facilmente integrado em aplicações Delphi já existentes podendo também ser utilizado em aplicações multidispositivo, como o FireMonkey. Além das empresas colherem os benefícios por meio da análise multidimensional dos dados de seus negócios, desenvolvedores de software podem adicionar em seus sistemas de informação transacionais módulos específicos para análise de dados. Isto aumentaria significativamente os benefícios e o valor agregado de um produto. Como visto nas primeiras seções, para que isso seja possível é necessário que uma base de dados transacional seja convertida em um d/w, de modo que os dados sejam preparados para permitir análises gerenciais eficientes. Além dos recursos básicos apresentados neste artigo, o *FastCube* possui outras funcionalidades mais avançadas, e alguns exemplos são: a utilização da linguagem *FastScript* para programação em *Object Pascal* dentro do cubo, a criação dos componentes via código e a importação de diversas fontes de dados para o mesmo cubo.

Autor

Jones Granatyr

Doutorando em Informática e Mestre em Ciência da Computação, ambos na área de Inteligência Artificial. Trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente. É professor da União de Ensino do Sudoeste do Paraná de Francisco Beltrão - PR.

Autor

Edson Emílio Scalabrin

Doutor em Ciência da Computação, tem experiência e realiza pesquisa na área, atuando principalmente nos seguintes temas: Agentes Autônomos, Modelagem de Sistemas Multiagente e Inteligência Artificial. É professor titular da Pontifícia Universidade Católica do Paraná.

Autor

Fábio Alexandre Taffe

Mestre em Ciência da Computação pela Universidade Federal de Santa Catarina, trabalha em projetos de pesquisa na área de Tecnologia da Informação pela União de Ensino do Sudoeste do Paraná – Unisep. É coordenador e professor do curso de Sistemas de Informação da Unisep de Francisco Beltrão – PR.

Autor

Alan Diego Darold

É acadêmico do curso de Sistemas de Informação da União de Ensino do Sudoeste do Paraná – Unisep – de Francisco Beltrão – PR. Trabalha em projetos comerciais de Business Intelligence e Data Wahouse.

Links:

FastCube
https://www.fast-report.com/en/product/fast-cube-2/

Fast Reports
https://www.fast-report.com

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Construindo Aplicativos Android no Delphi 10 Seattle

Desenvolvimento mobile com poucas linhas de código

Fique por Dentro

As últimas versões do Delphi têm acompanhado a evolução do mercado de aplicativos mobile, possibilitando o desenvolvimento de forma rápida - RAD (Rapid Application Development) para Windows, Android e iOS. Com o Delphi é possível desenvolver aplicativos multiplataforma, ou seja, com o mesmo código fonte é possível compilar o software para diversas plataformas e dispositivos móveis. Conhecer os recursos que a ferramenta disponibiliza para facilitar a criação de apps é importante para o desenvolvedor que deseja se manter atualizado e alcançar novos clientes com seus softwares, indo além do ambiente Windows, que tradicionalmente é o único sistema operacional alvo das aplicações Delphi.

Recentemente o Delphi passou de um IDE focado no Windows para tornar-se um que possibilita a construção de sistemas multiplataforma e também para dispositivos móveis. Essa mudança de perspectiva da Embarcadero com relação ao Delphi se deu principalmente devido ao grande crescimento dos dispositivos móveis, bem como dos sistemas operacionais Android e iOS. O Windows, que era considerado o sistema operacional mais popular, está agora atrás desses dois sistemas focados em dispositivos móveis. Os ambientes móveis possuem atualmente a liderança em termos de distribuição, sendo os mais comuns para necessidades de comunicação, jogos e entretenimento. Além disso, esses dispositivos estão se tornando cada vez mais populares para softwares empresariais, particularmente no que tange à integração com sistemas do tipo desktop já existentes em grande parte das empresas. Outra vertente nessa linha é o grande crescimento de uma revolução tecnológica chamada de Internet das Coisas, a qual possibilita que dispositivos dos mais variados tipos possam se comunicar e trocar informações. Vários autores consideram essa revolução como o futuro da computação e da comunicação.

Neste cenário, o Delphi possibilita uma rápida transição para que os desenvolvedores que já estão habituados com a programação baseada em componentes e/ou orientada a objetos possam criar aplicativos móveis. Neste sentido, o IDE apresenta um ambiente bastante familiar para aqueles que já conhecem o Delphi e desenvolvem aplicativos para Windows. A diferença básica é que para desenvolver aplicativos Windows é utilizada a VCL (*Visual Component Library*), enquanto que para desenvolver software multidispositivo as aplicações devem ser baseadas nos componentes da biblioteca FireMonkey (FMX - **BOX 1**), a qual gera código nativo e não depende das bibliotecas do Android ou iOS, por exemplo. Graças a essa biblioteca, o mesmo código fonte que rodará no Windows também funcionará em dispositivos móveis ou em qualquer outra plataforma em que o aplicativo for compilado. Esse fato torna o Delphi uma das ferramentas mais completas do mercado, pois

BOX 1. VCL x FireMonkey

A VCL, tradicional biblioteca do Delphi, encapsula a API nativa do Windows para prover uma série de funcionalidades e componentes visuais compatíveis com várias versões do sistema operacional, mesmo as mais antigas e ainda bastante utilizadas. Já o FireMonkey é uma camada de abstração da interface gráfica que internamente é compilada de forma nativa para diversas plataformas. Essa é atualmente a melhor opção para aplicações que farão uso de recursos gráficos em HD/3D, no entanto, seu objetivo não é a criação de games e sim aplicativos comerciais com interface rica e/ou mobile/multiplataforma.

 Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

um dos grandes problemas no desenvolvimento de aplicativos é tornar o código fonte independente de plataforma. Com o Delphi, nenhuma adaptação no código é necessária, ou seja, o aplicativo é programado uma única vez e pode ser instalado nas plataformas para as quais o Delphi oferece suporte. Com relação à Internet das Coisas, o Delphi também possibilita o desenvolvimento de aplicativos para os mais variados dispositivos, como o Google Glass.

Baseado neste cenário, o objetivo do presente artigo é apresentar como construir um aplicativo de cadastro no Delphi 10 Seattle, com um banco de dados local, instalando-o e testando-o em um dispositivo físico rodando o sistema operacional Android. Serão abordados recursos para realizar as operações básicas de cadastro, tais como inclusão, alteração, exclusão e navegação. Serão mostrados também dois recursos interessantes em aplicativos móveis, que são o compartilhamento dos dados do cadastro em redes sociais e a utilização do discador do aparelho para fazer ligações automaticamente de dentro da aplicação desenvolvida. Adicionalmente, será abordado como configurar estilos automáticos para melhorar o visual do aplicativo.

Primeiramente será mostrado como gerar uma base de dados utilizando o SQLite (**BOX 2**), a qual estará encapsulada dentro do próprio aplicativo e não será necessário criar um Web Service ou alguma conexão remota à fonte de dados. Posteriormente, será apresentado como utilizar o Delphi 10 Seattle para a criação da interface gráfica, bem como as operações para manipulação dos registros da base de dados e as configurações para que o aplicativo seja instalado no próprio dispositivo. Para finalizar, serão mostradas as operações de compartilhamento, a efetuação de ligações telefônicas e a configuração de estilos.

Criando o banco de dados SQLite no Delphi

A primeira etapa para a construção da base de dados é criar uma nova aplicação multidispositivo no Delphi 10 Seattle

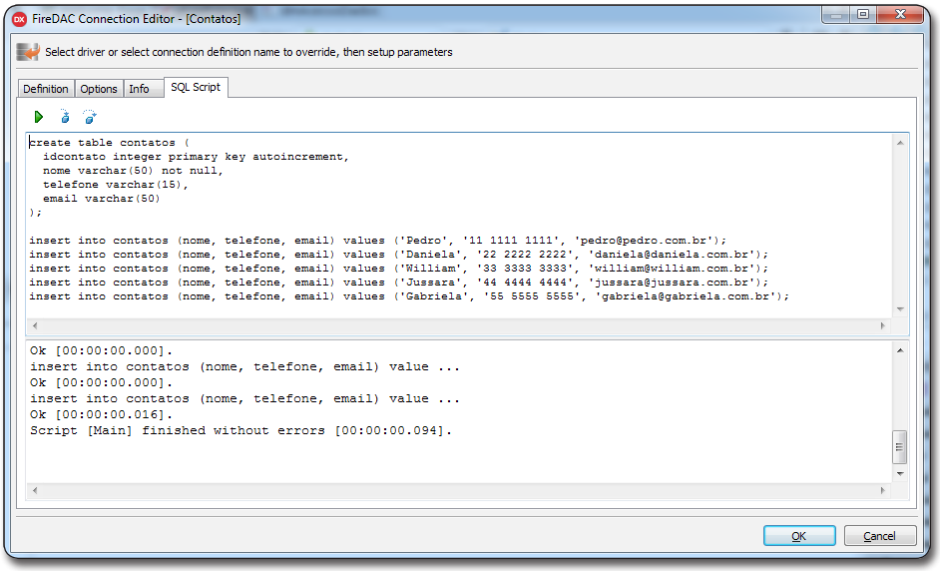


Figura 1. Criação do banco de dados

(*File>New>Multi-Device Application*) utilizando o layout em branco (*Blank Application*). Quando um novo projeto é criado seguindo esses passos, automaticamente o aplicativo estará utilizando a biblioteca de componentes FireMonkey, a qual garantirá a compilação das aplicações nativas para cada plataforma suportada. Nenhum plugin adicional precisa ser instalado no IDE para que os componentes do FireMonkey fiquem disponíveis.

A única tabela deste exemplo será construída diretamente pelo Delphi por meio do Data Explorer (*View>Data Explorer*). Com isso, não é necessária a utilização de outras ferramentas de acesso a dados para esse propósito, ou seja, todo o processo pode ser realizado dentro do próprio IDE.

Para criar um novo banco de dados pelo Data Explorer, primeiramente é necessário adicionar uma nova conexão ao FireDAC (*FireDAC>SQLite database>botão direito>Add New Connection*). Na janela *FireDAC Connection Editor* as únicas propriedades que devem ser configuradas é *Database* e *LockingMode*. Na primeira propriedade deve ser selecionado o caminho completo de onde o arquivo da base de dados será salvo, enquanto que a segunda propriedade deve ser marcada para *Normal*. A propriedade *Locking Mode* tem o objetivo de definir o bloqueio de conexão com o banco de dados, ou seja, no modo normal o arquivo do banco é liberado na conclusão de cada operação de leitura ou escrita. Por outro lado, no modo exclusivo o arquivo é bloqueado para gravação por um curto período de tempo. Para facilitar os testes, neste exemplo será utilizado o modo normal.

A **Figura 1** apresenta a guia *SQL Script* da janela *FireDAC Connection Editor*, juntamente com o comando SQL para a criação da tabela de contatos e os comandos para inserção de cinco registros para testes. O mesmo código apresentado na **Figura 1** pode ser observado na **Listagem 1**. Essa tabela é constituída por um campo chave primária auto incremento, o nome do contato, o telefone e o e-mail. Para certificar-se de que a tabela foi criada corretamente, basta executar um *Refresh* no nome da base de dados no *Data Explorer*.

BOX 2. SQLite

O SQLite é uma biblioteca escrita na linguagem C que implementa um banco de dados SQL, sem a necessidade de executar um serviço de sistema gerenciador de banco de dados ou outras dependências externas. É possível trabalhar com tabelas, índices, gatilhos ou visões, porém, não é recomendado para aplicações cliente/servidor ou que demandem muitos acessos e com grande quantidade de dados. Por esse motivo, o SQLite é bastante utilizado para aplicações em dispositivos móveis, mp3 ou gadgets. Em resumo, o SQLite é recomendado para softwares nos quais a base de dados estará encapsulada dentro do próprio aplicativo (embedded).

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 1. SQL para a criação da base de dados

```
01 create table contatos (
02  idcontato integer primary key autoincrement,
03  nome varchar(50) not null,
04  telefone varchar(15),
05  email varchar(50));

06 insert into contatos (nome, telefone, email) values
     ('Pedro', '11 1111 1111', 'pedro@pedro.com.br');
07 insert into contatos (nome, telefone, email) values
     ('Daniela', '22 2222 2222', 'daniela@daniela.com.br');
08 insert into contatos (nome, telefone, email) values
     ('William', '33 3333 3333', 'william@william.com.br');
09 insert into contatos (nome, telefone, email) values
     ('Jussara', '44 4444 4444', 'jussara@jussara.com.br');
10 insert into contatos (nome, telefone, email) values
     ('Gabriela', '55 5555 5555', 'gabriela@gabriela.com.br');
```

Criando o Data Module para acesso aos dados

O próximo passo do exemplo é a criação do Data Module para acesso aos dados (*File>New>Other>Delphi File>Data Module*), bem como a adição dos componentes que farão a conexão à base de dados e as operações na tabela de contatos. A **Figura 2** apresenta os componentes que devem ser adicionados no Data Module (dmAcessoDados), que são: TFDConnection (conexao), TFDQuery (consulta), TFDGUIxWaitCursor (cursor) e TFDPhysisSQLiteDriverLink (link). O componente FDConnection é responsável por realizar a conexão com a base de dados; clicando duas vezes sobre ele é aberta a janela de configuração, na qual a propriedade *Driver ID* deve ser configurada para *SQLite*, a propriedade *Database* deve apontar para o banco de dados criado na seção anterior, bem como a propriedade *LockingMode* deve ser marcada para *Normal*. Esses passos são os mesmos para a criação da base de dados vistos anteriormente. Adicionalmente, a propriedade *LoginPrompt* deve ser marcada para *False* para que a janela de usuário e senha não seja solicitada a cada nova interação com a base de dados. O componente FDQuery da biblioteca FireDAC é responsável pela consulta, inserção, edição e exclusão de registros da base de dados. Ele deve ter sua propriedade *Connection* ligada ao componente de conexão, enquanto que a propriedade *SQL* deve apresentar o seguinte comando para selecionar todos os registros e campos da tabela: `select * from contatos`. Os outros dois componentes não precisam ter nenhuma propriedade configurada, bastando que eles sejam adicionados ao Data Module. O FDGUIxWaitCursor permite o controle do cursor de espera, sendo necessário em aplicações baseados na biblioteca FireDAC. Por outro lado, o TFDPhysisSQLiteDriverLink representa o driver utilizado pelo SQLite, e cada banco de dados possui um componente específico.

Nota

É importante salientar que o "*" (asterisco) foi utilizado no comando SQL somente para fins deste exemplo, pois se a tabela contiver muitos registros o carregamento dos dados poderá ser lento, mesmo que a base de dados esteja armazenada localmente no dispositivo. Neste caso, recomenda-se a utilização de filtros para selecionar somente os registros necessários.

É necessário ainda abrir o *FieldsEditor* do componente FDConnection, acessando com clique do botão direito e escolhendo a opção *Add all fields* para que todos os campos sejam adicionados na consulta. Esse processo será útil nos próximos passos quando as configurações do formulário com LiveBindings forem executadas. A **Figura 2** mostra esse processo.

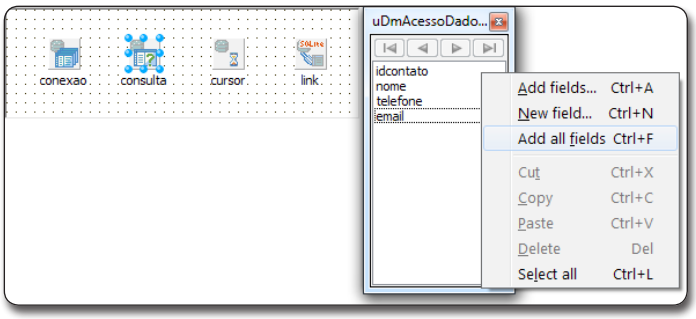


Figura 2. Data Module com os componentes de conexão

No evento *OnCreate* do Data Module, o seguinte código deve ser informado: consulta.Open(). Esse comando realiza a abertura da consulta, retornando para o componente FDQuery todos os registros cadastrados na base de dados.

Criando a interface gráfica

O exemplo é composto somente por um formulário (frmPrincipal), conforme mostrado na **Figura 3**. Ao topo encontra-se um componente TToolBar, que contém dez botões do tipo TButton que serão responsáveis pelas seguintes funções, respectivamente: anterior, próximo, incluir, alterar, excluir, gravar, cancelar, atualizar, compartilhar e fazer ligação. Para que os botões fiquem alinhados um ao lado do outro é necessário configurar a propriedade *Align* para o valor *Left*. O ícone é configurado por meio da propriedade *StyleLookup*, que disponibiliza uma lista de opções de botões decorados com ícones pré-definidos. Os seguintes estilos devem ser definidos para cada botão, respectivamente: *arrowlefttoolbutton*, *arrowrighttoolbutton*, *additembutton*, *composetoolbutton*, *trashtoolbutton*, *playtoolbutton*, *stoptoolbutton*, *refreshtoolbutton*, *actiontoolbutton* e *pagecurltoolbutton*. Na parte central do formulário devem ser adicionados três componentes TLabel (*lbNome*, *lbTelefone* e *lbEmail*) e três TEdits (*edtNome*, *edtTelefone* e *edtEmail*), conforme mostrado na **Figura 3**. Os componentes TBindSourceDB, TBindingsList, TStyleBook e TActionList (ações) serão explanados na sequência e não precisam ser adicionados ao formulário nesta etapa.

Para melhorar a interação do aplicativo com o usuário, o componente *edtTelefone* deve ter a sua propriedade *KeyboardType* configurada para *PhonePad*, enquanto que o componente *edtEmail* deve ter essa propriedade setada para *EmailAddress*. Com isso, quando o aplicativo for executado no aparelho e o usuário tocar nesses componentes, será aberto um teclado numérico para informar o número do telefone e um teclado com opções para digitar o email, respectivamente. Essa opção é bastante útil por dois motivos. O primeiro é para limitar o usuário somente com os caracteres permitidos por cada campo, evitando que sejam

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

informados dados inválidos. O segundo motivo é que em aplicativos móveis a resolução em geral é pequena, portanto, não faria sentido apresentar um teclado completo para o campo telefone, visto que ele aceita somente números. Além das opções de teclado para telefone e email, é também possível definir teclados somente com letras, números e URLs. Na **Figura 3** é possível observar um exemplo de teclado específico para números de telefone, o qual poderá ser visualizado somente quando o aplicativo for instalado no dispositivo móvel.

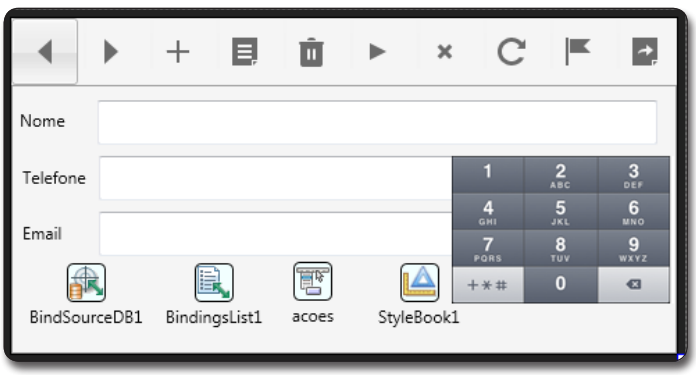


Figura 3. Interface gráfica do aplicativo

A **Figura 4** mostra o combobox para a visualização de estilos, que pode ser configurado para as seguintes plataformas: Windows, OSX, iOS e Android. Na esquerda é possível observar como a barra de ferramentas e os botões ficarão nas três últimas plataformas, respectivamente. É importante salientar que essa opção é somente uma configuração para visualização da aplicação em cada plataforma, e nenhuma codificação é gerada no projeto para alteração de estilos quando o aplicativo for executado no dispositivo móvel.

Manipulando os registros da base de dados

As ligações dos componentes da interface gráfica com a base de dados serão feitas por meio do recurso de Live Bindings, o qual permite a ligação dos controles das janelas diretamente com os registros da base de dados existentes no componente FDQuery. Para isso, primeiramente é necessário adicionar a unit do Data Module no formulário principal (*File>Use Unit*) e logo após clicar com o botão direito em qualquer parte do formulário e selecionar a opção Bind Visually. A **Figura 5** apresenta o esquema de ligação da propriedade *Text* dos componentes TEdits com cada campo da consulta.

Nota

Alguns componentes do FireMonkey, tais como os TEdits, disponibilizam a propriedade ControlType, a qual especifica o tipo de controle. Caso um componente seja configurado com o valor Styled, a renderização do componente no dispositivo móvel será feita de acordo com as bibliotecas nativas do FireMonkey. Por outro lado, se o componente for configurado com o valor Platform o mesmo será renderizado de acordo com o estilo nativo da plataforma na qual a aplicação será instalada. Essa configuração é semelhante à divisão de TControl entre TGraphicControl e TWindowsControl na VCL.

Nota

Na **Figura 4**, ao lado da opção Style encontra-se outro combobox chamado View, o qual apresenta opções de configurações para diversos tipos de aparelhos e dimensões, tais como iPad, iPhone, Android Tablet, Android Phone e Google Glass. Essa funcionalidade permite a personalização de estilos próprios para cada plataforma, ou seja, no iPhone 3.5" pode-se desejar que os botões fiquem alinhados à esquerda, enquanto que no Android Tablet 10" deseja-se que os botões fiquem alinhados à direita. A visão Master é a genérica que funcionará em todos os dispositivos e personalizações podem ser feitas a partir dela. Para isso, são utilizados mecanismos de reaproveitamento de código utilizando Orientação a Objetos. Quando utilizadas as propriedades de alinhamento dos componentes do FireMonkey, o próprio IDE trata de gerenciar os posicionamentos nos mais variados dispositivos.

Pode-se notar nessa figura que o campo chave primária (idcontato) não foi ligado com nenhum componente visual da tela. Isso se deve ao fato de que na definição da tabela este campo foi definido como auto incremento e seu valor será gerado automaticamente pelo SQLite. Após ter realizado as ligações, dois novos componentes foram automaticamente adicionados no formulário: TBindingsList e TBindSourceDB, conforme pode ser observado na **Figura 3**. O primeiro é responsável por manter a lista de todas as ligações disponíveis na aplicação, enquanto que o segundo tem a função de tornar os dados contidos em uma fonte de dados disponíveis para os componentes de tela que o acessarem. Pode-se notar que a propriedade *DataSet* deste componente está ligada ao componente de consulta aos dados (FDQuery), o que indica a ligação com todos os campos da tabela de contatos.

As operações de manipulação da base de dados serão feitas sem nenhuma codificação utilizando o componente TActionList (ações), que pode ser observado na **Figura 3** e é responsável por manter a lista de todas as ações do aplicativo. Para isso serão adicionadas ações padrões, as quais podem ser configuradas dando duplo clique no componente e selecionando a opção *New Standard Action*. Serão escolhidas dez ações, uma correspondente a cada botão adicionado na barra de ferramentas superior, que

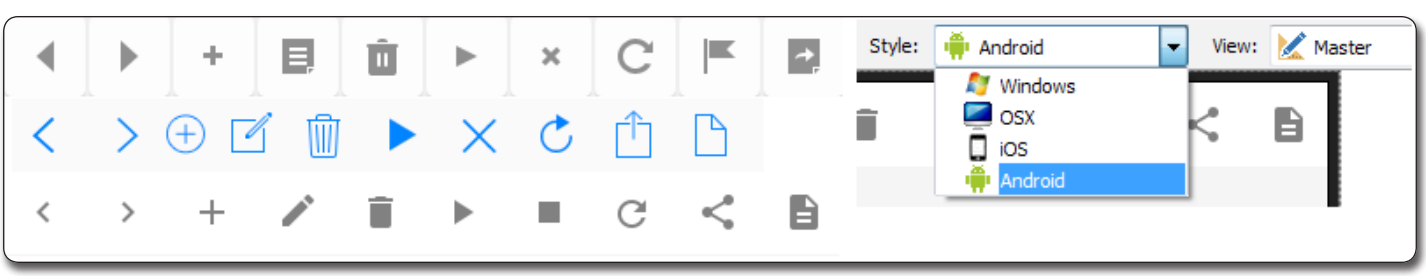


Figura 4. Visualização de estilos

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

são: *BindNavigatePrior* (anterior), *BindNavigateNext* (próximo), *BindNavigateInsert* (incluir), *BindNavigateEdit* (alterar), *BindNavigateDelete* (excluir), *BindNavigatePost* (gravar), *BindNavigateCancel* (cancelar), *BindNavigateRefresh* (atualizar), *ShowShareSheetAction* e *PhoneCallAction*. As oito primeiras ações são relativas à manipulação básica dos registros da base de dados, enquanto que as duas últimas dizem respeito ao compartilhamento de dados com mídias sociais e realizar ligações telefônicas diretamente pelo aplicativo, respectivamente. Essas duas últimas ações serão abordadas em detalhes na sequência.

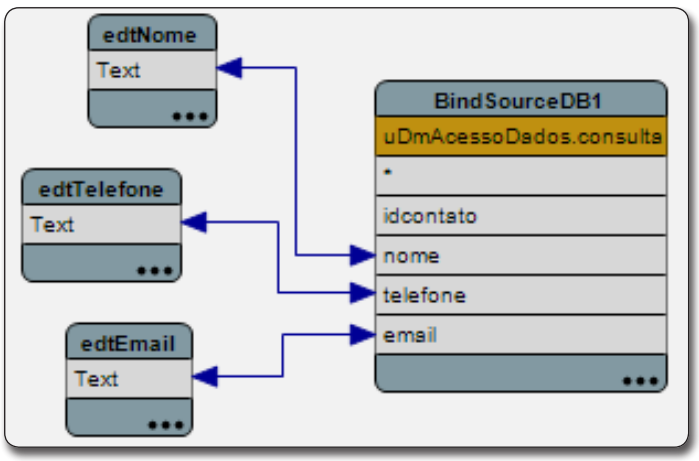


Figura 5. Ligação dos componentes com Live Bindings

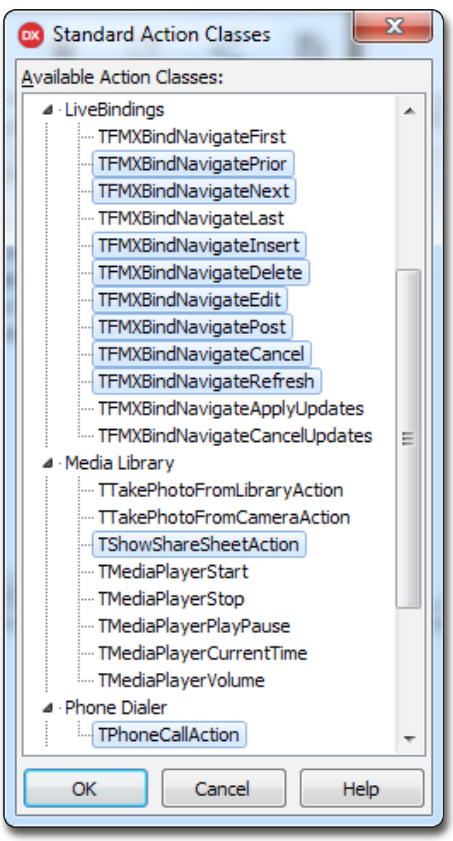


Figura 6. Seleção de ações no ActionList

A **Figura 6** apresenta a janela para a escolha das ações no ActionList, que estão divididas em três categorias: Live Bindings, Media Library e Phone Dialer. Pode-se notar que na categoria Media Library existem outras opções, tais como: tirar foto diretamente da câmera, selecionar foto da galeria e ações relativas ao controle do player. Neste artigo serão mostradas as ações *ShowShareSheetAction* e *PhoneCallAction*.

Para que as operações básicas de manipulação de dados possam funcionar, é necessário que todas as ações da categoria Live Bindings tenham sua propriedade *DataSource* ligada ao componente *BindSourceDB1*. Para finalizar, os oito primeiros botões da barra de ferramentas superior devem ter a sua propriedade *Action* configurada para cada uma das ações correspondentes adicionadas anteriormente. Esse processo fará com que as ações sejam disparadas a cada toque nos botões.

Com isso, o aplicativo já está funcional e com todas as operações básicas de manipulação de bases de dados concluída. Como pôde ser observado, a programação para dispositivos móveis no RAD Studio 10 Seattle é exatamente igual a programar um aplicativo tradicional Win32 / VCL, ou seja, praticamente não existe curva de aprendizado para quem já trabalha com aplicativos Windows e deseja desenvolver aplicativos móveis ou multiplataforma. Caso essa aplicação seja executada, será aberta uma janela no padrão do Windows com todas as operações funcionando. Na próxima seção será apresentado como esse mesmo aplicativo pode ser executado diretamente de um smartphone físico com o Android.

Configurando o dispositivo físico para executar as aplicações Delphi

O RAD Studio apresenta um emulador para executar as aplicações Android, porém, ele não é o ideal em termos de desempenho e responsividade para a depuração e os testes dos aplicativos. Essa é uma limitação do SDK do Android e não do RAD Studio, portanto, a maneira mais indicada para testar os aplicativos é utilizar um dispositivo físico Android compatível com os aplicativos gerados pelo Delphi, podendo ser smartphone ou tablet. É suportada a maioria dos dispositivos Android modernos, porém, nem todos os modelos de aparelhos e versões do Android funcionam, principalmente os mais antigos. Visite a seção **Links** para consultar uma lista da Embarcadero e obter maiores detalhes sobre as versões suportadas e não suportadas.

O próximo passo para configurar o dispositivo é instalar o seu driver USB correspondente no mesmo computador onde o RAD Studio está instalado. Normalmente este é fornecido pelo próprio fabricante do aparelho, porém, é importante que seja feito o download e instalação do driver com a versão correta do fabricante do modelo, caso contrário, o dispositivo não será reconhecido. Após ter sido instalado é necessário que o *modo desenvolvedor* e *depuração USB* estejam ativos no aparelho Android. Essa ativação varia de dispositivo para dispositivo e na seção **Links** encontra-se um tutorial básico sobre como realizar essa operação.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

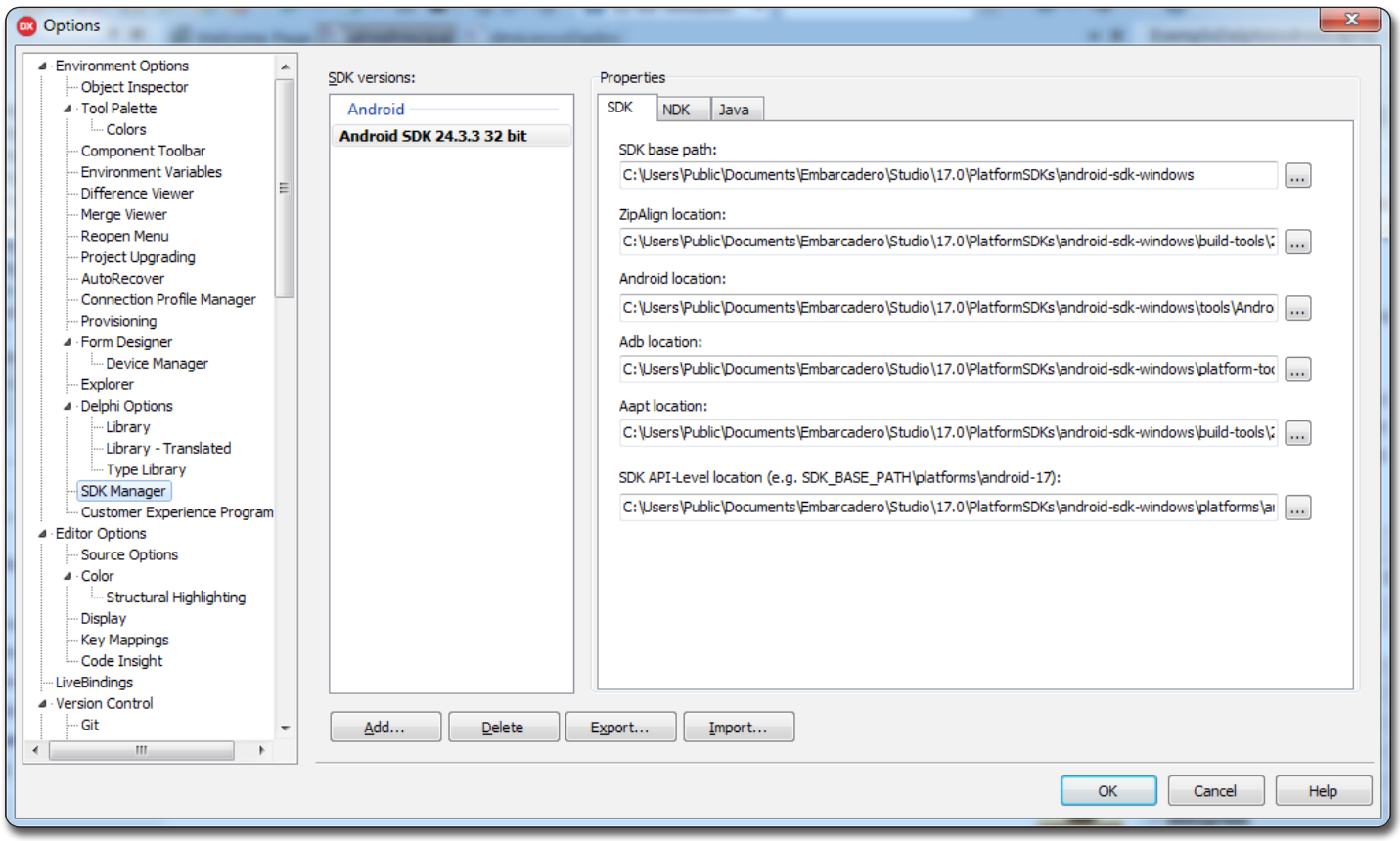


Figura 7. Configuração do SDK Manager no RAD Studio

Fazendo o deploy da aplicação para um celular com Android

Antes de fazer o deploy do aplicativo é necessário verificar se o SDK, NDK e Java estão configurados corretamente no computador, uma vez que esses componentes formam a base do ambiente necessário para o desenolvimento de aplicações Android. Para isso, deve ser aberta a janela Options (menu *Tools>Options*) e no menu à esquerda deve ser selecionada a opção *SDK Manager*, conforme mostrado na **Figura 7**. Pode-se notar que nesta janela existem três guias (SDK, NDK e Java), nas quais os caminhos devem estar corretamente configurados. É importante salientar que essas configurações já devem ter sido feitas automaticamente durante a instalação do Delphi 10 Seattle.

O próximo passo é ligar o dispositivo físico no computador utilizando o cabo USB. Neste momento, se tudo estiver configurado corretamente, o nome do modelo do aparelho deverá ser listado pelo RAD Studio na janela *Project Manager* em *Target Platforms (Android)*. É possível observar na **Figura 8** que foi adicionado o aparelho LG-D175f, que foi utilizado para os testes deste artigo. Caso o modelo do aparelho não apareça, as configurações apresentadas anteriormente devem ser revistas. Na seção **Links** estão disponíveis maiores informações sobre os dispositivos suportados, bem como configurar o IDE para detectar os dispositivos Android.

Outra opção que deve ser alterada são as configurações da construção do aplicativo em *Build Configurations*, existindo as opções *Debug* e *Release*. Caso a opção debug seja marcada, será possível realizar depurações diretamente pelo dispositivo físico. Por outro lado, a opção release deve ser escolhida quando não for necessário fazer depuração no próprio dispositivo, sendo geralmente utilizada quando a versão final do aplicativo for instalada no aparelho. Neste exemplo, a opção *Release* será utilizada, pois o tamanho do pacote e o tempo de implantação são menores. Na **Figura 8** é também possível observar que podem ser configuradas as outras plataformas para as quais o RAD Studio pode compilar os aplicativos, tais como Windows 32 bits ou 64 bits, iOS 32 bits ou 64 bits, iOS Simulator e OS X. Por meio dessas opções é possível compilar os softwares desenvolvidos no RAD Studio para múltiplas plataformas. Com isso, pode-se escrever uma única versão do código fonte do aplicativo, compilá-lo e distribuí-lo para todas essas plataformas, o que diminui drasticamente o tempo de migração do software para dispositivos distintos. Vale ressaltar que se quiser rodar este aplicativo em um Windows Desktop, por exemplo, não é necessário criar um novo projeto. Em versões anteriores do IDE ainda havia essa divisão Desktop / Mobile, mesmo com FireMonkey. Nas últimas versões do RAD Studio, devido ao recurso conhecido como FireUI (**BOX 3**), o mesmo projeto pode ser desenhado e compilado para qualquer plataforma, seja ela Desktop ou Mobile, reaproveitando-se código .PAS e .FMX (equivalente ao DFM da VCL).

É necessário ainda adicionar o arquivo da base de dados ao projeto (*Project>Add to Project*), selecionando o *contatos.sdb* que foi criado na primeira etapa. Acessando a opção *Project> Deployment* é possível observar os arquivos que serão carregados

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

para o aparelho quando o aplicativo for instalado, e dentre eles, encontra-se na lista o arquivo *contatos.sdb*, conforme apresentado na **Figura 9**. O *Remote Path* foi configurado automaticamente para o caminho *.\assets\internal*, que indica que este arquivo será armazenado neste local do dispositivo Android. Ainda na **Figura 9** pode-se notar que foi desmarcarda a opção *gdbserver*, que é um programa para realizar depuração remota e o mesmo não será instalado no dispositivo por não ser necessários no exemplo construído.

BOX 3. FireUI

Inserido no RAD Studio na versão XE7, o FireUI é um conjunto de ferramentas que permite o desenvolvimento de interfaces gráficas para diversas plataformas e resoluções de tela. Com isso o IDE conta com componentes visuais, propriedades e opções no editor para compor as interfaces de forma que elas se adaptem corretamente a smartphones, tablets e PCs.

Nota

Para executar o deploy de uma aplicação que suporte o iOS é necessário possuir uma instalação do sistema operacional Mac OS X e ter uma conta de desenvolvedor na Apple. Consulte a seção **Links** para mais informações.

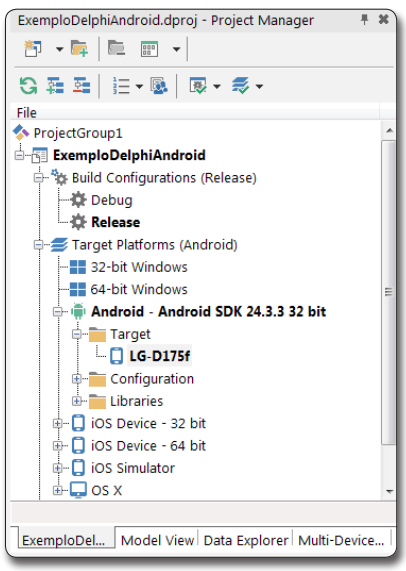


Figura 8. Configurações para o deploy

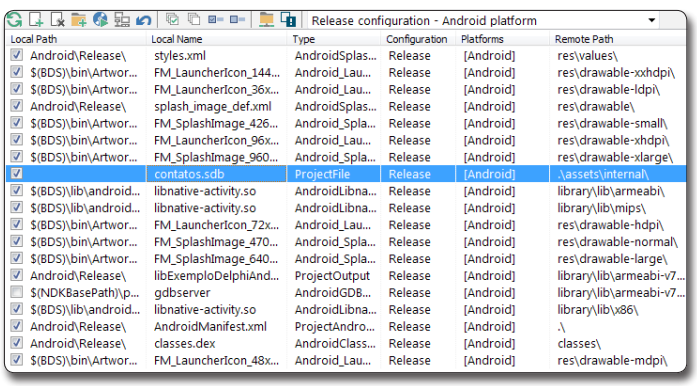


Figura 9. Arquivos para o release da aplicação

Por fim, é preciso realizar a configuração do caminho da base de dados em tempo de execução no evento *BeforeConnect* do componente FDConnection que está localizado no Data Module. A **Listagem 2** apresenta o código correspondente.

Listagem 2. Configuração do acesso à base de dados.

```
01 uses System.IOUtils;
02 procedure TuDmAcessoDados.conexaoBeforeConnect(Sender: TObject);
03 begin
04   conexao.Params.Values['Database'] := TPath.GetDocumentsPath +
       PathDelim + 'contatos.sdb';
05 end;
```

A linha 1 importa a biblioteca IOUtils para serem realizadas operações de acesso aos diretórios do aparelho, enquanto que a linha 4 configura o parâmetro Database do componente FDConnection para receber o endereço do arquivo da base de dados. O código *PathDelim* é utilizado para garantir que o caminho retornado esteja com o delimitador à direita "/".

A partir deste momento, a aplicação já pode ser executada e o RAD Studio fará automaticamente o deploy para o hardware físico. Esse processo pode ser rápido ou levar alguns minutos até ser concluído, dependendo da configuração do computador e do dispositivo móvel escolhido. Quando finalizado, a aplicação abrirá automaticamente no aparelho e todas as operações de manipulação da base de dados podem ser testadas assim como foi feita na versão para Windows.

Compartilhando dados e fazendo ligações telefônicas

O penúltimo botão que foi adicionado na barra de ferramentas tem a função de compartilhar em mídias sociais ou outros meios os dados do contato selecionado. No componente ActionList (ações) adicionado anteriormente já foi criada a ação padrão *Show-ShareSheetAction*, bastando agora que a propriedade *Action* deste botão seja configurada para estar ligada com esta ação. Para que isso seja possível é necessário digitar o código da **Listagem 3** no evento *BeforeExecute* da ação *ShowSharedSheetAction*.

Como pode ser observado, a propriedade *TextMessage* da ação recebe a concatenação do nome, do telefone e do e-mail do contato. Quando esse evento é disparado é aberta uma janela de compartilhamento no dispositivo móvel, na qual é possível compartilhar esses dados com diversas mídias sociais, tais como e-mail, WhatsApp, Facebook, dentre outros. Essa funcionalidade é bastante útil quando se deseja compartilhar todos os dados de um cadastro de forma automática sem ter a necessidade de copiar e colar os valores de cada campo, sendo semelhante à opção para compartilhar contato existente no WhatsApp.

Para realizar ligações telefônicas diretamente pelo aplicativo foi adicionado anteriormente no ActionList a ação padrão *PhoneCallAction*, bastando que a propriedade *Action* do último botão da barra de ferramentas seja configurada para esta ação.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Figura 10. Editor de estilos do FireMonkey

O código apresentado na **Listagem 4** foi programado no evento *Update* da ação *PhoneCallAction*, o qual atribui o número do telefone que está contido no TEdit para a propriedade TelephoneNumber. Desta forma, o discador do aparelho é aberto automaticamente e já realiza a ligação para o contato.

Nota

Se quiser executar essa aplicação com o mesmo código fonte para dispositivos móveis e desktop, poderá utilizar diretivas de compilação para que a linha 4 não seja incluída caso a compilação seja feita em um dispositivo Windows, por exemplo.

Listagem 3. Compartilhamento dos dados do contato.

```
01 procedure TfrmPrincipal.ShowShareSheetAction1BeforeExecute
     (Sender: TObject);
02 begin
03   ShowShareSheetAction1.TextMessage := 'Nome: ' + edtNome.Text +
       'Telefone: ' + edtTelefone.Text + ' Email: ' + edtEmail.Text;
04 end;
```

Listagem 4. Efetuando ligações.

```
01 procedure TfrmPrincipal.PhoneCallAction1Update(Sender: TObject);
02 begin
03   PhoneCallAction1.TelephoneNumber := edtTelefone.Text;
04 end;
```

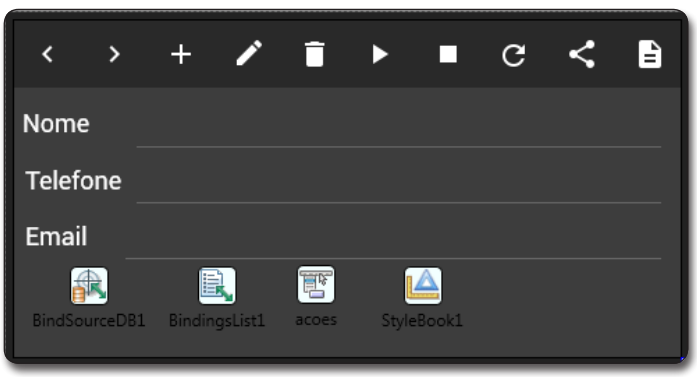


Figura 11. Estilo aplicado no aplicativo

Nota

A Embarcadero disponibiliza um pacote chamado FireMonkey Premium Styles Pack, o qual possui um conjunto maior de estilos pré-definidos. Além disso, por meio do FireMonkey Style Designer é possível criar estilos personalizados, assim os desenvolvedores possuem uma maior gama de ferramentas para criarem seus próprios estilos.

Configurando estilos

O FireMonkey disponibiliza estilos diferenciados para a personalização das aplicações para dispositivos móveis, os quais podem ser encontrados no seguinte diretório: C:\Users\Public\Documents\Embarcadero\Studio\17.0\Styles\Android. Por padrão, no RAD Studio 10 Seattle estão disponíveis seis diferentes estilos, que são: AndroidDark, AndroidLDark, AndroidDarkBlue, AndroidLight, AndroidLLight e GoogleGlass. Para configurá-los na aplicação é necessário adicionar o componente TStyleBook no formulário principal, conforme apresentado na **Figura 3**. O formulário principal (frmPrincipal) deve ter sua propriedade *StyleBook* ligada a este novo componente, para que os estilos possam ser aplicados. Duplo clique no StyleBook abre o editor de estilos do FireMonkey, conforme mostrado na **Figura 10**.

Clicando no primeiro botão é aberta uma janela onde se pode selecionar os estilos previamente instalados que estão disponíveis no diretório informado anteriormente. Neste exemplo será selecionado o estilo *AndroidDark* e após deve ser clicado no botão *Apply and Close* para que o estilo seja aplicado em todo o aplicativo. A **Figura 11** apresenta o aplicativo com o novo visual.

A vantagem de utilizar estilos é que o processo para configuração do visual do aplicativo fica mais fácil e rápido, sem a necessidade de ter que configurar todos os componentes individualmente. É importante salientar que diferentemente do combobox *Style* apresentado na **Figura 4**, os estilos definidos desta forma serão enviados para o dispositivo móvel quando o aplicativo for instalado.

A **Figura 12** apresenta o aplicativo sendo executado diretamente pelo dispositivo. Na esquerda é mostrada a aplicação original e no meio é apresentado o teclado numérico, o qual foi configurado anteriormente por meio da propriedade *KeyBoardType*.

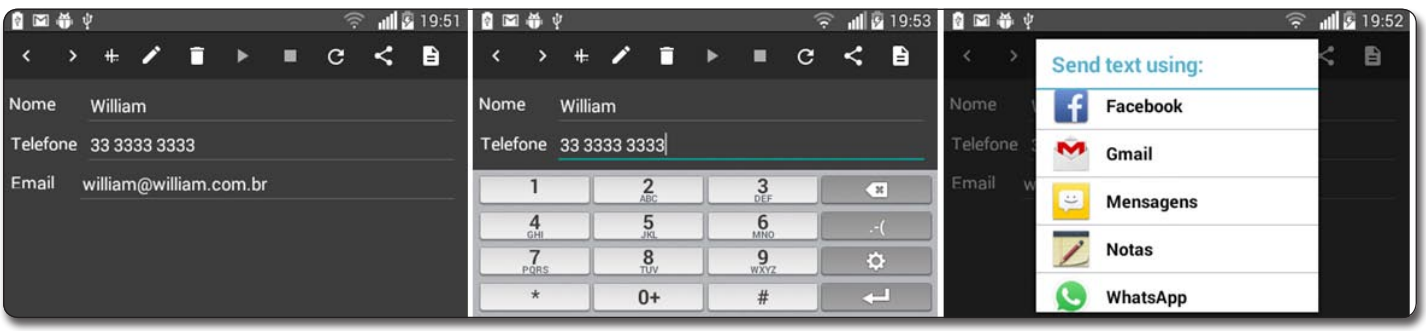


Figura 12. Estilo aplicado no aplicativo

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Por fim, a imagem da direita mostra a janela de compartilhamento do contato.

O Delphi, juntamente com a biblioteca FireMonkey, apresentam um conjunto de componentes que agilizam o desenvolvimento de aplicações nativas para dispositivos móveis ou outras plataformas. A grande vantagem do Delphi é que desde as primeiras versões foi utilizado o conceito de RAD (*Rapid Application Development*) para o rápido desenvolvimento de software. Nas últimas versões do Delphi esse conceito foi transportado para o desenvolvimento de aplicativos móveis, o que o torna atualmente uma das ferramentas mais produtivas para o desenvolvimento para móveis. Outra grande vantagem é o fato de poder compilar os aplicativos para uma diversa gama de plataformas, o que torna o desenvolvimento ainda mais rápido e sem a necessidade de adaptações no código fonte. Neste artigo foi apresentado o desenvolvimento de um pequeno aplicativo para cadastro de contatos, juntamente com alguns recursos básicos para compartilhamento de dados e a realização de ligações telefônicas diretamente pelo sistema. Além desses recursos básicos apresentados neste artigo, o FireMonkey possui uma gama de componentes para as mais variadas funções, tais como: listagem de dados em forma de grade, acesso a câmera e ao flash do dispositivo, diversos sensores, notificações, gerenciamento do media player, dentre outros. A utilização desses recursos possibilita o desenvolvimento de aplicativos móveis, multiplataforma e baseados nos conceitos de Internet das Coisas, com uma grande gama de recursos.

Autor

Jones Granatyr

Doutorando em Informática e Mestre em Ciência da Computação, ambos na área de Inteligência Artificial. Trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente. É professor da União de Ensino do Sudoeste do Paraná de Francisco Beltrão - PR.

Autor

Guinther Pauli

Doutorando em Informática e Mestre em Ciência da Computação, ambos na área de Inteligência Artificial. Trabalha em projetos de pesquisa relacionados à área de Inteligência Artificial, tais como Sistemas Especialistas, Mineração de Dados, Mineração de Textos e Sistemas Multiagente. É professor da União de Ensino do Sudoeste do Paraná de Francisco Beltrão - PR.

Autor

Thiago Luiz Lauxen

É acadêmico do curso de Sistemas de Informação da União de Ensino do Sudoeste do Paraná – Unisep – de Francisco Beltrão – PR. Trabalha em empresa de desenvolvimente de software utilizando o Delphi.

Autor

Jean Paul Barddal

Bacharel em Ciência da Computação e Mestre em Informática pela Pontifícia Universidade Católica do Paraná (PUCPR). Atualmente é doutorando em Informática pela mesma instituição, atuando na área de mineração de dados e aprendizagem de máquina, majoritariamente a partir de fluxos contínuos de dados.

Autor

Fábio Spak

Graduado em Ciência da Computação e membro do grupo de pesquisa Inteligência Computacional na Universidade do Contestado - SC no campus de Porto União. Já atuou como professor de informática e atualmente trabalha em ambiente administrativo e de suporte à tecnologia da informação.

Autor

Fábio Alexandre Taffe

Mestre em Ciência da Computação pela Universidade Federal de Santa Catarina. Trabalha em projetos de pesquisa na área de Tecnologia da Informação pela União de Ensino do Sudoeste do Paraná – Unisep. É coordenador e professor do curso de Sistemas de Informação da Unisep de Francisco Beltrão – PR.

Links e Referências:

Site do SQLite
http://www.sqlite.org

Documentação da Embarcadero
http://docwiki.embarcadero.com

Dispositivos e versões do Android suportados pelo Delphi
http://docwiki.embarcadero.com/RADStudio/XE8/en/Android_Devices_Supported_for_Application_Development

Ativação do modo desenvolvedor e depuração USB
http://www.techtudo.com.br/dicas-e-tutoriais/noticia/2014/10/como-ativar-o-modo-desenvolvedor-no-android.html

Configurando seu sistema para detectar o seu dispositivo Android
http://docwiki.embarcadero.com/RADStudio/XE8/en/Configuring_Your_System_to_Detect_Your_Android_Device

DUARTE, William. Delphi para Android e iOS: Desenvolvendo Aplicativos Móveis. Rio de Janeiro: Brasport, 2015.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

REVISTAS DIGITAIS

Imagine poder ter acesso a todas as edições das revistas Java Magazine, .NET Magazine, SQL Magazine, Mobile Magazine, Engenharia de Software, ClubeDelphi e Easy Java.

São mais de 4.000 artigos publicados!

Uma verdadeira biblioteca online perfeita para seus estudos!

Para mais informações :
http://www.devmedia.com.br/mvp

DEVMEDIA

Automatizando a atualização do banco de dados em Delphi

Atualizando a base através de um dicionário de dados

O gerenciamento de scripts que realizam a atualização do banco de dados no cliente, em geral, gera problemas. Primeiramente por que constantemente temos scripts que por algum motivo não foram executados e ocasionam erros no cliente, os quais geralmente conseguimos identificar pela mensagem e dizem respeito à falta de um campo que o script não criou ou atualizou. Outra dificuldade ocorre por que em algum momento o cliente pode ter perdido algumas atualizações do sistema e quando se tenta executar o script da versão X até a Y com frequência temos problemas. Além disso, é difícil efetuar o controle de versão desses scripts, pois não temos a possibilidade de rollback a menos que criemos um script separado para isso, sem contar que esses arquivos, com passar do tempo, vão ficando cada vez maiores. Como resolver esse problema? Qual seria a melhor solução para isso? Uma alternativa de resolução para essa questão envolve a utilização de dicionários de dados, que serão explicados a seguir.

Dicionários e o banco de dados

O dicionário, nesse contexto, é uma estrutura de dados que contém a representação dos elementos do banco de dados. No contexto dos SGDBs, esses dicionários representam a estrutura do nosso banco, contendo informações como descrição dos objetos, restrições de integridade, stored procedures, triggers, índices, chaves e validações.

O uso de dicionários para representação da base de dados traz algumas vantagens que incluem maior facilidade para atualização do banco e para integração deste com a aplicação, que consegue mais facilmente reconhecer a estrutura das tabelas e exibir uma interface adequada para o usuário.

Fique por Dentro

Neste artigo veremos como efetuar atualizações de banco de dados utilizando apenas uma estrutura que define como o banco de dados deve ser. Este conceito, denominado de dicionário de dados, visa substituir a utilização dos scripts de atualização das estruturas do banco de dados no cliente por uma solução mais eficiente. Essa solução irá nos garantir que a estrutura atual do banco está de acordo com os dados do dicionário, independente de quantas e quais versões o cliente deixou de atualizar.

Neste caso teremos um arquivo com todas as definições do banco de dados e esse arquivo vai ser definido manualmente por nós, então podemos acrescentar nele algumas informações que poderão ser importadas e aproveitadas na aplicação. Uma dessas informações cuja definição podemos otimizar é o *caption* da coluna, que geralmente atribuímos manualmente a cada label nas telas do sistema. Utilizando a estratégia do dicionário podemos importar esse *caption* diretamente do arquivo de definição do banco de dados, com isso teremos o mesmo texto sendo exibido em qualquer tela do sistema onde aquele campo for utilizado. De forma análoga podemos atribuir descrições para os campos das tabelas e exibir essas informações como *hint* nos componentes.

A seguir temos detalhados alguns detalhes sobre como o uso de dicionários pode facilitar o desenvolvimento e manutenção de sistemas em Delphi.

Para o banco de dados:

• Garantia de que a estrutura esteja condizente com a realidade: o dicionário, neste caso, será um espelho de como o banco de dados deverá ficar;

• Arquivo de "script" fica menor: como não teremos mais arquivos de scripts, o arquivo de dicionário os substituirá. Esse arquivo, a longo prazo, será proporcionalmente menor que os scripts devido à não redundância das informações. Além disso, armazenar

 Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

a estrutura de uma tabela é mais simples que armazenar seus comandos SQL equivalentes;
• Possibilidade de *rollback*: como o banco será estruturado de acordo com o dicionário, e este contém apenas a estrutura e não comandos SQL, fica fácil pegar a estrutura do dicionário e aplicar no banco, então para efetuar o *rollback*, basta utilizar a versão anterior do dicionário.

Para o sistema:
• Podemos utilizar o dicionário para definir se um campo é obrigatório, não tendo mais necessidade de deixar isso fixo no código. Em caso de um campo se tornar obrigatório ou não no dicionário, isso irá refletir no sistema sem codificação e na maioria das implementações sem a necessidade de rodar uma nova versão, tendo que apenas reenviar o dicionário para o cliente;
• *Captions* e *Hints* refletem no sistema sem a necessidade de códigos engessados. Assim como nos campos obrigatórios, em caso alterações não necessitamos de uma nova versão;
• Podemos criar domínios e validações exclusivas para um determinado grupo de colunas e empregá-las apenas informando no dicionário. Um exemplo para isso seria a máscara de um telefone;
• Podemos definir a ordem de apresentação das colunas pelo dicionário, retirando essa tarefa do programador e deixando centralizado no dicionário, isso torna a alteração muito mais fácil;
• O sistema conhece completamente a estrutura do banco de dados.

Criando o Dicionário

A partir de agora vamos criar nosso dicionário de dados e utilizá-lo em uma aplicação. Para isso, precisamos primeiramente definir o que desejamos armazenar no dicionário, que serão basicamente as informações que definem as tabelas, colunas e índices do banco de dados. Nas **Tabelas 1** a **3** vemos quais dados serão salvos no arquivo e seu significado para a aplicação.

Essas informações serão armazenadas em um arquivo XML, um formato simples e que pode ser facilmente manipulado em aplicações Delphi através do componente TXMLDocument, que se encontra na paleta Internet. Na **Listagem 1** vemos como deve ficar a estrutura do arquivo XML do dicionário, com os elementos e atributos equivalentes aos que vimos nas tabelas.

A estrutura desse template é bem simples. Dentro da tag *Tabelas* temos uma outra para cada tabela do banco, que por sua vez contém tags internas para os campos e os índices. Os dados vistos nas tabelas anteriores são adicionados como atributos dentro de cada tag.

Gerando as classes para leitura e escrita do XML

Esse é um processo bem simples, visto que toda a inteligência ficará por conta do Delphi. Vamos criar um projeto em branco e adicionar em um form o componente *TXMLDocument* que se encontra na paleta Internet, como pode ser visto na **Figura 1**.

Dados da Tabela	
Dado	**Descrição**
Nome	Representa o nome da tabela
Descricao	Descrição da tabela

Tabela 1. Dados da tabela

Dados das colunas	
Dado	**Descrição**
Ordem	Nos informa em qual ordem o campo deve ser apresentado no grid
Nome	Representa o nome da coluna
Descricao	Descrição da coluna
Caption	Texto que deve ser apresentado para essa coluna
Hint	Hint que deve se apresentado para essa coluna
Tipo	Tipo de dados no banco de dados
Tamanho	Tamanho da coluna
Decimal	Caso seja valor numérico, utilizar para casas decimais
Obrigatorio	Define se a coluna é de preenchimento obrigatório
AutoIncremetar	Define se a coluna é auto incremental
Único	Define se os valores para essa coluna são únicos
Valor_Defaut	Valor padrão para a coluna
Visivel	Define se esse campo vai ser visível no sistema

Tabela 2. Dados das colunas

Dados dos índices	
Dado	**Descrição**
Nome	Nome do índice
Campos	Campos que compõem o índice

Tabela 3. Dados dos índices

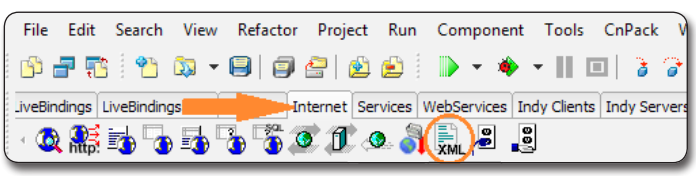


Figura 1. Componente TXMLDocument na paleta Internet

Com o componente adicionado ao formulário, vamos dar dois cliques nele e será aberta uma janela de wizard. Na primeira tela (**Figura 2**) será solicitado o caminho do nosso arquivo XML, depois de informa-lo basta clicar em "Next".

Na tela seguinte as tags do arquivo XML serão reconhecidas automaticamente pelo Delphi, como vemos na **Figura 3**, e serão gerados tipos de dados equivalentes a cada uma delas.

Expandindo os itens do treeview do lado esquerdo podemos ver as propriedades de cada tipo (**Figura 4**), que são gerados automaticamente a partir dos atributos das tags do XML.

Lendo o dicionário pelo Delphi

Para que possamos ler o dicionário, vamos agora criar nossas classes que farão acesso à classe de leitura do XML, facilitando

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 1. Estrutura do arquivo XML do dicionário

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Metadata do Banco de Dados -->
<DicionarioDeDados Version="1.0">
  <Tabelas>
    <Tabela Nome="Tabela1" Descricao="Descrição dessa tabela">
      <Campos>
        <Campo Ordem="1"
          Nome="campo1"
          Descricao="Desc Campo 1"
          Caption="Capt Campo 1"
          Hint="Hint Campo 1"
          Tipo="Integer"
          Tamanho="10"
          Decimal="0"
          Obrigatorio="S"
          AutoIncremetar="S"
          Unico=""
          Valor_Defaut=""
          Visivel="N"
        />
        <Campo Ordem="2"
          Nome="campo2"
          Descricao="Desc Campo 2"
          Caption="Capt Campo 2"
          Hint="Hint Campo 2"
          Tipo="varchar"
          Tamanho="12"
          Decimal=""
          Obrigatorio="S"
          AutoIncremetar="N"
          Unico=""
          Valor_Defaut=""
          Visivel="S"
        />
      </Campos>
      <Indices>
        <Indice Nome="primary" Campos="campo1"/>
        <Indice Nome="Idx_1"  Campos="campo2"/>
      </Indices>
    </Tabela>
  </Tabelas>
</DicionarioDeDados>
```

o acesso e manipulação do nosso dicionário. Na **Figura 5** podemos analisar o diagrama das nossas classes de acesso aos dados do arquivo.

Para agrupar os dados que são comuns às três principais estruturas do nosso dicionário criaremos a classe *TItem*, cujo código pode ser visto na **Listagem 2** e da qual herdarão as classes *TColuna*, *TTabela* e *TItemIndice*, que representam os elementos armazenados no arquivo XML.

Listagem 2. Unit com implementação da classe TItem

```
unit uItem;

interface
Uses Classes;
Type
  TItem = Class(TCollectionItem)
    private
      FNome     : String;
      FDescricao : String;
    published
      Property Nome     : String read FNome     write FNome     ;
      Property Descricao : String read FDescricao write FDescricao ;
  end;

implementation

end.
```

Além delas, teremos uma classe referente à coleção de cada um dos itens, ou seja, as classes *TCollectionColuna*, *TCollectionTabela* e *TCollectionItemIndice*. Por fim temos a classe *TDicDados*, que representa o dicionário como um todo e possui métodos para carregar os dados, bem como um atributo do tipo *TCollecionTabela* indicando as tabelas do banco de dados.

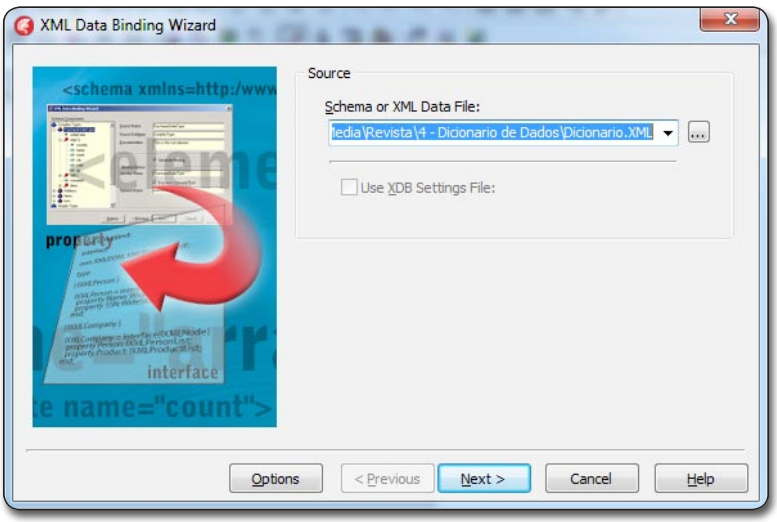


Figura 2. XML Data Binding Wizard - informar source

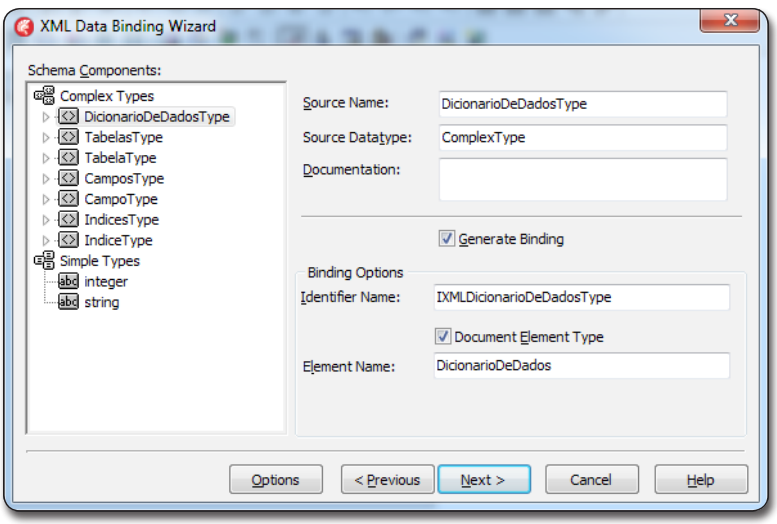


Figura 3. XML Data Binding Wizard Schema Components

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

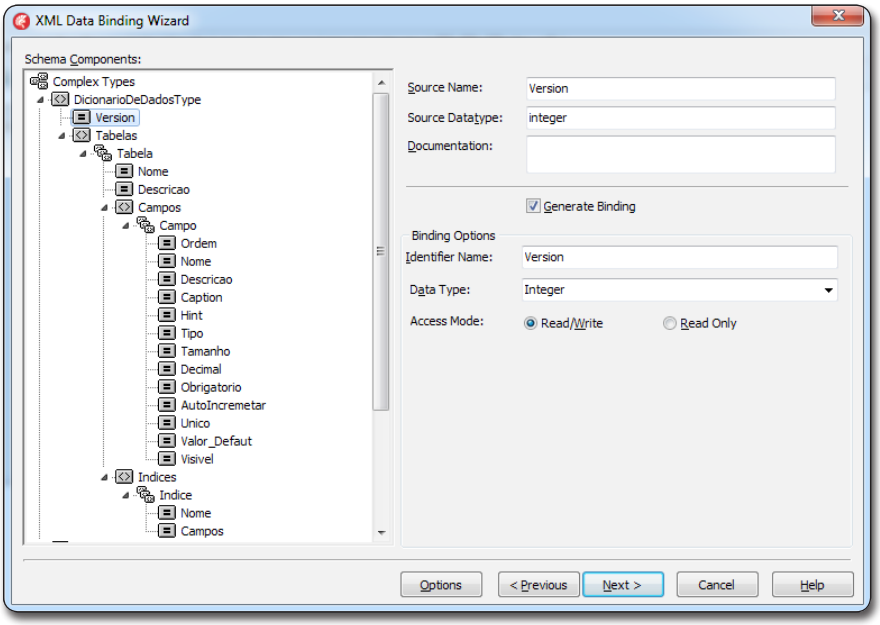


Figura 4. XML Data Binding Wizard Schema Components com arvore aberta

A classe *TItem* apenas herda da classe *TCollectionItem* (linha 6) já presente na unit *Classes* (linha 4) e contém duas propriedades (nome e descrição) que serão comuns a todas as classes de item do dicionário, o que facilitará futuras implementações. Nas **Listagens 3** a **5** temos o código das classes que herdam de *TItem*.

Listagem 3. Unit com implementação da classe TItemIndice

```
01 unit uItemIndice;
02
03  interface
04  Uses uItem;
05
06  Type
07   TItemIndice = Class(TItem)
08    private
09     FCampos : String;
10    published
11     Property Campos : String read FCampos  write FCampos ;
12   end;
13
14  implementation
15
16  end.
```

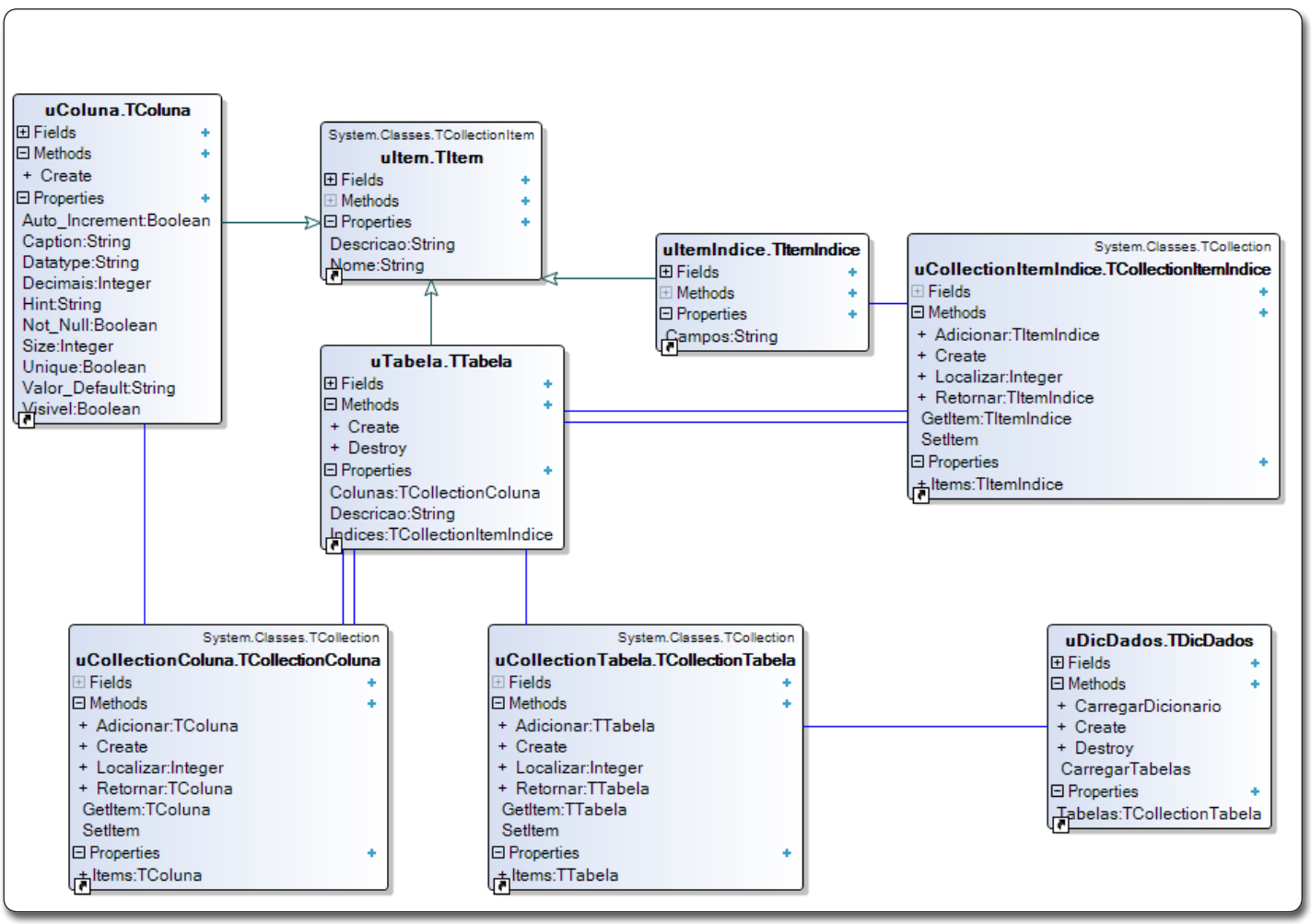


Figura 5. Diagrama de classes do dicionário

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 4. Unit com implementação da classe TColuna

```
01  unit uColuna;
02
03  interface
04  Uses uItem, Classes;
05
06  Type
07   TColuna = Class(TItem)
08    private
09    FCaption      : String;
10     FHint        : String;
11     FDatatype    : String;
12     FSize        : Integer;
13     FDecimais    : Integer;
14     FAuto_Increment : Boolean;
15     FNot_Null    : Boolean;
16     FUnique      : Boolean;
17     FValor_Default  : String;
18     FVisivel     : Boolean;
19    public
20     constructor Create(Collection: TCollection); override;
21    published
22     Property Caption : String read FCaption write FCaption;
23     Property Hint : String read FHint write FHint;
24     Property Datatype : String read FDatatype write FDatatype;
25     Property Size : Integer read FSize write FSize;
26     Property Decimais : Integer read FDecimais write FDecimais;
27     Property Auto_Increment : Boolean read  FAuto_Increment write
                                               FAuto_Increment;
28     Property Not_Null : Boolean read FNot_Null write FNot_Null;
29     Property Unique : Boolean read FUnique write FUnique ;
30     Property Valor_Default : String read FValor_Default write
                                               FValor_Default ;
31     Property Visivel : Boolean read FVisivel write FVisivel;
32   end;
33
34  implementation
35
36  { TColuna }
37
38  constructor TColuna.Create(Collection: TCollection);
39  begin
40   inherited;
41   FCaption          := '';
42   FHint             := '';
43   FDatatype         := '';
44   FSize             := 0 ;
45   FDecimais         := 0 ;
46   FAuto_Increment := False;
47   FNot_Null         := False;
48   FUnique           := False;
49   FValor_Default    := '';
50   FVisivel          := False;
51  end;
52
53  end.
```

Listagem 5. Unit com implementação da classe TTabela

```
01  unit uTabela;
02
03  interface
04  Uses uItem, uCollectionItemIndice,Classes,
05  uColuna,uItemIndice, SysUtils,uCollectionColuna;
06
07  Type
08   TTabela = Class(TItem)
09    private
10     FColunas   : TCollectionColuna;
11     FIndices   : TCollectionItemIndice;
12    public
13     constructor Create(Collection: TCollection); override;
14     destructor Destroy; override;
15    published
16     Property Colunas   : TCollectionColuna      read FColunas;
17     Property Indices   : TCollectionItemIndice read FIndices;
18   end;
19
20  implementation
21
22  { TTabela }
23  constructor TTabela.Create(Collection: TCollection);
24  begin
25   inherited;
26   FColunas  := TCollectionColuna.Create(Nil);
27   FIndices  := TCollectionItemIndice.Create(Nil);
28  end;
29
30  destructor TTabela.Destroy;
31  begin
32   FreeAndNil(FColunas);
33   FreeAndNil(FIndices);
34   inherited;
35  end;
36
37  end.
```

Essa classe basicamente possui uma propriedade do tipo string que deve conter os nomes dos campos envolvidos no índice (linhas 9 e 11).

A classe *TColuna* é a que possui maior número de propriedades, cujos valores são iniciados como valores default no construtor, de forma a evitar erros devidos ao não preenchimento dos valores.

Como já herdamos de *TItem*, a classe *TTabela* implementa adicionalmente apenas duas propriedades referentes às coleções de colunas e índices (linhas 10, 11, 16 e 17). A classe sobrescreve os métodos do construtor e destrutor devido à necessidade de instanciar as coleções de colunas e índices, removendo-as da memória posteriormente.

Para fazer a leitura do dicionário temos a classe *TDicDados*, cujo código do protótipo pode ser visto na **Listagem 6**.

Nessa classe foram definidas duas propriedades: uma é a coleção de tabelas do dicionário (linha 8) e a outra é o caminho do arquivo (linha 7). Na linha 13 declaramos o construtor da classe, cuja implementação encontra-se na **Listagem 7** e é responsável por iniciar os valores das propriedades. Já o destrutor (linha 14) simplesmente libera da memória essas propriedades, por isso foi omitido deste artigo.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

O método *CarregarDicionario*, que pode ser visto na **Listagem 8**, é onde validamos a existência do arquivo XML do dicionário (linhas 3 a 8) e em caso positivo chamamos o método *CarregarTabelas* (linha 11), que pode ser visto na **Listagem 9**.

Listagem 6. Protótipo da classe TDicDados

```
01  Uses uCollectionTabela, uTabela, SysUtils, Dialogs, Forms, Dicionario,
02  uColuna,uItemIndice, uCollectionColuna ;
03
04  Type
05   TDicDados = Class
06    private
07     FFile   : String;
08     FTabelas : TCollectionTabela;
09
10    protected
11     Procedure CarregarTabelas;
12    public
13     Constructor Create;
14     Destructor  Destroy;
15
16     Procedure CarregarDicionario;
17    published
18     property Tabelas : TCollectionTabela read FTabelas;
19   End;
```

Listagem 7. Create do TDicDados

```
01 constructor TDicDados.Create;
02 begin
03  FFile   := ExtractFilePath(Application.ExeName) + '\Dicionario.XML';
04  FTabelas := TCollectionTabela.Create(Nil);
05 end;
```

Listagem 8. Método CarregarDicionario da TDicDados

```
01 procedure TDicDados.CarregarDicionario;
02 begin
03  If Not(FileExists(FFile)) then
04  Begin
05   MessageDlg('Não foi possivel localizar o dicionário',
06              mtError, [mbOK], 0);
07   Exit;
08  end;
09
10 Try
11  CarregarTabelas;
12 except
13  on e : Exception do
14   begin
15    MessageDlg('Ocorreu um erro ao atualizar as tabelas.'
16           + #13#10+ E.Message,  mtError, [mbOK], 0);
17    Exit;
18   end;
19 End;
20 end;
```

Nesse método declaramos as variáveis de uso da interface de acesso ao XML e seus respectivos elementos, depois temos os itens para manipular os objetos de tabela, coluna e índice e duas variáveis de iteração (linhas 3 a 8). Na linha 10 carregamos o arquivo XML para a variável *sFile*, em seguida fazemos um laço para todos os elementos referentes a tabelas do arquivo, adicionando mais um item de tabela à coleção, atribuindo os valores de nome e descrição recuperados do arquivo. Um segundo laço (linha 22) é feito para todos os elementos correspondentes às colunas da tabela e para cada coluna localizada é instanciado um objeto e adicionado à tabela com seus respectivos valores atributos preenchidos (linhas 24 a 40). Na sequência (linhas 44 a 52) é efetuado um laço sobre os elementos de índices com a mesma intenção do laço nas colunas.

Para testar as classes e seu funcionamento vamos criar um botão e um memo no formulário principal, onde vamos instanciar um objeto de *TDicDados* e acionar seu método *CarregarDicionario*.

Listagem 9. Método CarregaTabelas da TDicDados

```
01  procedure TDicDados.CarregarTabelas;
02  Var
03   sFile      : IXMLDicionarioDeDadosType;
04   iTabela    : IXMLTabelaType;
05   Tabela     : TTabela;
06   Coluna     : TColuna;
07   Indice     : TItemIndice;
08   i,j        : Integer;
09  begin
10   sFile := LoadDicionarioDeDados(FFile);
11
12   for i := 0 to sFile.Tabelas.Count - 1 do
13   Begin
14    iTabela  := sFile.Tabelas.Tabela[i];
15
16    //Dados da Tabela tabela
17    Tabela              := Tabelas.Adicionar;
18    Tabela.Nome         := iTabela.Nome;
19    Tabela.Descricao    := iTabela.Descricao;
20
21    //Começa a gravar os campos dessa tabela
22    for j:=0 to sFile.Tabelas.Tabela[i].Campos.Count -1 do
23    Begin
24     Coluna := Tabela.Colunas.Adicionar;
25
26     With sFile.Tabelas.Tabela[i].Campos.Campo[j] do
27     Begin
28      Coluna.Nome              := Nome;
29      Coluna.Descricao         := Descricao;
30      Coluna.Caption           := Caption;
31      Coluna.Hint              := Hint;
32      Coluna.Datatype          := Tipo;
33      Coluna.Size              := Tamanho;
34      Coluna.Decimais          := Decimal;
35      Coluna.Not_Null          := Obrigatorio   = 'S';
36      Coluna.Auto_Increment := AutoIncremetar = 'S';
37      Coluna.Unique            := Unico       = 'S';
38      Coluna.Valor_Default     := Valor_Defaut;
39      Coluna.Visivel           := Visivel = 'S';
40     end;
41    end;
42
43    //Começa a gravar os Indices dessa tabela
44    for j:=0 to sFile.Tabelas.Tabela[i].Indices.Count -1 do
45    Begin
46     Indice    := TItemIndice(Tabela.Indices.Adicionar);
47     With sFile.Tabelas.Tabela[i].Indices.Indice[j] do
48     Begin
49      Indice.Nome  := Nome;
50      Indice.Campos := Campos;
51     End;
52    end;
53   End;
54  end;
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Depois disso vamos percorrer o dicionário e adicionar seus itens ao memo. Faremos isso apenas para testar a funcionalidade antes de ir para a próxima fase. O código do teste ficará como na **Listagem 10** e se todos os passos foram seguidos e tudo ocorreu bem, você terá um resultado semelhante ao da **Figura 6**.

Listagem 10. Método CarregaTabelas da TDicDados

```
01  procedure TfrmMain.Button1Click(Sender: TObject);
02  Var DicDados : TDicDados;
03     i,j : Integer;
04  begin
05    DicDados := TDicDados.Create;
06    DicDados.CarregarDicionario;
07
08    mmo1.Clear;
09    for I := 0 to DicDados.Tabelas.Count-1 do
10    Begin
11      mmo1.Lines.Add(DicDados.Tabelas.Items[i].Nome);
12      for j := 0 to DicDados.Tabelas.Items[i].Colunas.Count -1 do
13        mmo1.Lines.Add(DicDados.Tabelas.Items[i].Colunas.Items[j].Nome);
14
15      for j := 0 to DicDados.Tabelas.Items[i].Indices.Count -1 do
16        mmo1.Lines.Add(DicDados.Tabelas.Items[i].Indices.Items[j].Nome);
17    End;
18  end;
```

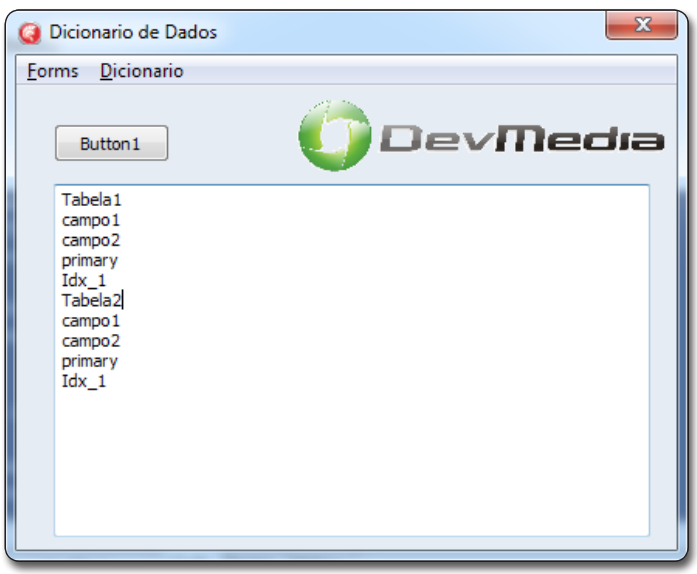


Figura 6. Teste de funcionalidade do dicionario

Na linha 5 criamos um objeto do tipo *TDicDados* e na linha 6 chamamos o método de carregar o dicionário que vai popular o objeto. Em seguida efetuamos um laço na coleção de colunas (linhas 12 e 13) e adicionamos cada uma ao memo, fazendo depois o mesmo procedimento com os índices (linhas 15 e 16).

Agora com o dicionário pronto faremos a comparação com a estrutura do banco de dados e sua atualização.

Comparando e atualizando a estrutura do banco de dados

Para essa fase vamos utilizar o banco de dados MySQL e os componentes de conexão e consulta do FireDac.

Criaremos no projeto um form (**Figura 7**) que será chamado do menu e será responsável por efetuar as comparações de estrutura e atualizações no nosso banco de dados. Esse formulário contém uma barra de progresso para manter o usuário atualizado sobreo andamento do processo, um memo para exibir o log de operações, e dois botões, um para executar a comparação e outro para fechar nosso formulário.

No botão de fechar basta colocar o modal result como mrClose ou apenas chamar o método *Close* para fechar o formulário. No botão Executar vamos adicionar algumas chamadas aos métodos que serão responsáveis pela comparação da estrutura, como vemos nas linhas 7 e 8 da **Listagem 11**.

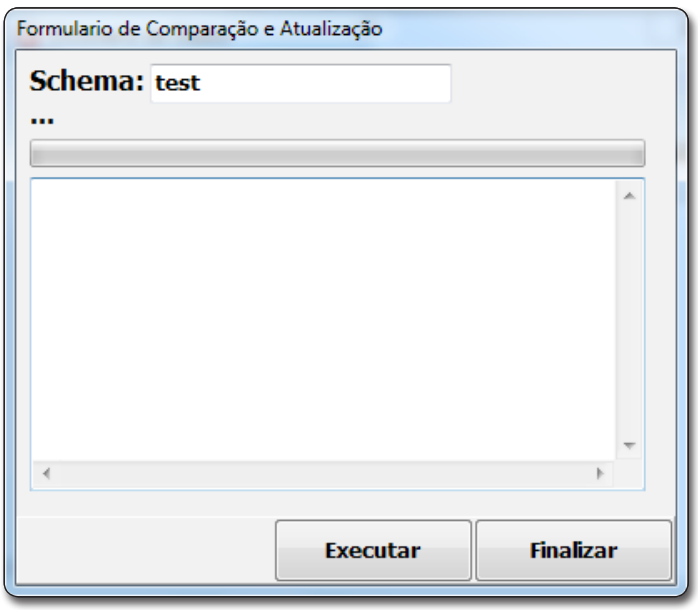


Figura 7. Formulário de atualização

Listagem 11. Método do click do botão executar

```
01 procedure TfrmComparaDB.btn_ExecutarClick(Sender: TObject);
02 begin
03   btn_Finalizar.Enabled := False;
04   btn_Executar.Enabled  := False;
05
06   try
07     MontarSQLs;
08     Comparar;
09   finally
10     btn_Finalizar.Enabled := True;
11     btn_Executar.Enabled  := True;
12   end;
13 end;
```

Nesse método desabilitamos os botões de finalizar e executar para evitar possíveis problemas na interação com o usuário e no finally habilitamos novamente. O primeiro método a ser executado é o *MontarSQLs* (**Listagem 12**), responsável por preparar nossas queries com os comandos SQL que farão a coleta das informações do banco de dados. Já o *Comparar* é responsável pela comparação da estrutura.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Os métodos *MontarSQLTabelas*, *MontarSQLColunas* e *MontarSQLIndices*, utilizados nas linhas 3 a 5 são responsáveis por compor as instruções SQL que recuperarão os dados das tabelas, colunas e índices do banco de dados. Nas **Listagens 13** a **15** temos as instruções utilizadas nesses métodos.

Listagem 12. Método MontarSQLs

```
01 procedure TfrmComparaDB.MontarSQLs;
02 begin
03   MontarSQLTabelas;
04   MontarSQLColunas;
05   MontarSQLIndices;
06 end;
```

Listagem 13. SQL da consulta de tabelas

```
01 SELECT
02   TABLE_SCHEMA,
03   TABLE_NAME
04 FROM INFORMATION_SCHEMA.TABLES
05 WHERE TABLE_SCHEMA = :TABLE_SCHEMA
06 ORDER BY TABLE_NAME DESC;
```

Listagem 14. SQL da consulta de colunas

```
01   SELECT
02     TABLE_SCHEMA,
03     TABLE_NAME ,
04     COLUMN_NAME,
05     DATA_TYPE,
06     CASE IS_NULLABLE
07       WHEN "NO" THEN
08         True
09       ELSE
10         False
11     END NOT_NULL ,
12     CASE
13       WHEN CHARACTER_MAXIMUM_LENGTH IS NOT NULL THEN
14         CHARACTER_MAXIMUM_LENGTH
15       ELSE
16         NUMERIC_PRECISION
17     END SIZE ,
18     NUMERIC_SCALE,
19     CASE EXTRA
20       WHEN "auto_increment" THEN
21         true
22       ELSE
23         false
24     END AUTO_INCREMENT
25   FROM INFORMATION_SCHEMA.COLUMNS
26   WHERE TABLE_SCHEMA = :TABLE_SCHEMA
27   AND   TABLE_NAME       = :TABLE_NAME
28   ORDER BY TABLE_NAME DESC
```

Nessa instrução efetuamos um select sobre a tabela *TABLES* do *INFORMATION_SCHEMA* do MySQL e adicionamos uma condição para que sejam retornados apenas os registros correspondentes ao schema que vai ser passado por parâmetro.

A instrução da **Listagem 14** retorna mais dados e é um pouco mais complexa devido a algumas validações necessárias. Ela efetua um select na tabela *COLUMS* com duas condições: uma é o schema e o outro é referente à tabela da qual desejamos obter essas colunas. Tivemos também que adicionar um case à coluna *IS_NULLABLE* devido ao seu retorno ser uma string ("NO" ou "YES") e para facilitar a manipulação dessa informação no código devemos retornar esse dado como booleano. Verificamos se ele não é nulo e retornamos o valor em booleano com o alias NOT_NULL, que será verdadeiro se o campo for marcado como não nulo, e falso em caso contrário. O próximo case é feito para pegar o tamanho do campo. Como o MySQL separa essas informações dependendo do tipo de campo e na nossa estrutura isso não é aplicado, temos que retornar esses dados de acordo com o nosso padrão. No MySQL, para os tipos numéricos a informação do tamanho do campo fica na coluna *NUMERIC_PRECISION*, já para os demais tipos essa informação fica em CHARACTER_MAXIMUM_LENGTH, o que foi feito nesse case foi validar se existe valor em uma coluna ou na outra e retorná-lo sob o alias *SIZE*. O último case é para saber se a coluna é auto incremento e essa informação fica como uma string na coluna *EXTRA*.

Listagem 15. SQL da consulta de Índices

```
01 SELECT
02   INDEX_NAME,
03   COLUMN_NAME
04 FROM INFORMATION_SCHEMA.STATISTICS
05 WHERE TABLE_SCHEMA = :TABLE_SCHEMA
06 AND   TABLE_NAME       = :TABLE_NAME
```

Nossa última consulta é responsável por retornar os índices de uma tabela específica de um certo schema, ambos passados como parâmetro para uma consulta na tabela *STATISTICS*.

Nossas consultas estão determinadas e agora é hora de fazer a comparação, então vamos analisar o método que será responsável por essa tarefa, o método *Comparar* visto na **Listagem 16**.

Nesse método instanciamos o dicionário na variável *DicDados* (linha 7) e em seguida (linhas 10 e 11) passamos para o parâmetro da query de tabelas o nome do schema do banco de dados, valor contido no edit *edt_schema*, e em executamos a consulta. Na linha 16 chamamos o método *CarregarDicionario* e depois fazemos um laço sobre ele para comparar com a atual estrutura do banco de dados. Nesse trecho temos algumas interações com a barra de progresso e com o memo que está registrando os eventos. Na linha 28 é validado se existe uma tabela no banco com o nome da que temos no dicionário, se não existir, devemos criar (*CreateTable*) e caso já exista devemos verificar ser existe alguma alteração a ser feita (*AlterTable*). Passamos para esses métodos a tabela que desejamos verificar e eles retornam uma string que é adicionada ao log. Para entender como a criação e alteração das tabelas funciona, vamos analisar primeiramente o método *CreateTable* na **Listagem 17**.

Esse método é responsável pela criação da estrutura de uma tabela no banco de dados, porém ele apenas prepara a instrução SQL responsável por essa tarefa e envia para o método *ExcutarScript*. Temos ainda a chamada aos métodos *CreateColunas* e *CreateIndices*, que retornam o trecho do script responsável pela criação das colunas e índices, respectivamente. Na **Listagem 18** vemos o código do método *CreateColunas*.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 16. Método Comparar

```
01  procedure TfrmComparaDB.Comparar;
02  Var DicDados : TDicDados;
03    i              : Integer;
04    Tabela       : TTabela;
05  begin
06
07   DicDados := TDicDados.Create;
08   Try
09    QryTabelas.Close;
10    QryTabelas.ParamByName('TABLE_SCHEMA').AsString:= edt_Schema.Text;
11    QryTabelas.Open();
12    try
13     mmoLog.Clear;
14
15     mmoLog.Lines.Add('Carregando Dicionario');
16     DicDados.CarregarDicionario;
17
18     pb.Position := 0;
19     pb.Max      := DicDados.Tabelas.Count;
20
21     mmoLog.Lines.Add('Comparando estruturas');
22     for I := 0 to DicDados.Tabelas.Count-1 do
23     Begin
24      pb.Position := i+1;
25
26      Tabela := DicDados.Tabelas.Items[i];
27      mmoLog.Lines.Add('Comparando Tabela -> ' + Tabela.Nome);
28      If Not(QryTabelas.Locate('TABLE_NAME',
          Tabela.Nome,[loCaseInsensitive])) Then
29       mmoLog.Lines.Add(CreateTable(Tabela))
30      else
31       mmoLog.Lines.Add(AlterTable(Tabela))
32     End;
33    Except
34     on e:exception do
35      mmoLog.Lines.Add('Erro ao comparar estruturas:' + e.Message)
36    end;
37   finally
38    FreeAndNil(DicDados);
39   End;
40  end;
```

Listagem 17. Método CreateTable

```
01 function TfrmComparaDB.CreateTable(pTabela: TTabela): String;
02 Var
03  SQL : String;
04 begin
05  Result := '';
06
07  SQL := 'Create Table '+edt_Schema.Text+'.'+pTabela.Nome + '(';
08  SQL := SQL + CreateColunas(pTabela);
09  SQL := SQL + CreateIndices(pTabela);
10  SQL := SQL + ')';
11
12  ExecutarScript(SQL);
13 end;
```

Esse método faz um loop sobre as colunas da tabela que foi passada com parâmetro e monta uma instrução SQL com os dados encontrados no dicionário. Com isso obtemos uma string com os dados da coluna a ser adicionada em nosso script.

Na **Listagem 19** temos o método *CreateIndices*, de função semelhante, mas que gera o trecho do script responsável por criar os índices da tabela.

Listagem 18. Método CreateColunas

```
01 function TfrmComparaDB.CreateColunas(pTabela: TTabela): String;
02 Var
03  i    : Integer;
04  Coluna : TColuna;
05 Begin
06  Result := '';
07  for i := 0 to pTabela.Colunas.Count -1 do
08  Begin
09   Coluna := pTabela.Colunas.Items[i];
10   mmoLog.Lines.Add('Comparando Coluna -> ' + Coluna.Nome);
11   if (i > 0) Then
12    Result := Result + ',';
13
14   Result := Result + #13#10 +
15          Coluna.Nome + ' ' +
16          Coluna.Datatype + '(' + Coluna.Size.ToString+'' +
17          iif(Coluna.Decimais > 0,
             ','+Coluna.Decimais.ToString,'') +')'+ ' '+
18          iif(Coluna.Not_Null,'Not null','') + ' ' +
19          iif(Coluna.Auto_Increment,'auto_increment','');
20  End;
21 end;
```

Listagem 19. Método CreateIndices

```
01 function TfrmComparaDB.CreateIndices(pTabela: TTabela): String;
02 Var
03  Indice : TItemIndice;
04  i    : Integer;
05 begin
06  Result := '';
07  for i    := 0 to pTabela.Indices.Count -1 do
08  Begin
09   Indice := pTabela.Indices[i];
10   mmoLog.Lines.Add('Comparando Indice ->' + Indice.Nome);
11   if (UpperCase(Indice.Nome) <> 'PRIMARY') then
12    begin
13     Result := Result + #13#10+
14           ', INDEX '+ Indice.Nome + '(' + Indice.Campos +')';
15    end
16   else
17    begin
18     Result := Result + #13#10+
19           ', PRIMARY KEY ('+ Indice.Campos +')';
20    end;
21  end;
22 end;
```

Nota

O método iif foi criado para simular o operador ternário, encontrado em outras linguagens, e economizar linhas de código com a redução dos ifs utilizados.

Aqui fazemos um loop na lista de índices da tabela no dicionário e validamos se o índice é *primary key* ou não. Essa validação é necessária por que a criação do índice para *Index* e para *Primary Key* são um pouco diferentes, como vemos nas linhas 11 a 21.

Vejamos agora o método *ExecutarScript*, utilizado dentro do *CreateTable* para executar as instruções geradas pelos métodos que acabamos de ver. Seu código encontra-se na **Listagem 20.**

Aqui basicamente passamos a string recebida como parâmetro para a propriedade SQL da query e executamos com o método *ExecSQL*, retornando uma mensagem para ser gravada no log.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Com isso finalizamos a parte de criação das estruturas do banco, vamos agora partir para o método *AlterTable*, contido na **Listagem 21**.

Listagem 20. Método ExecutarScript

```
01 function TfrmComparaDB.ExecutarScript(pScript: String): String;
02 begin
03  Result := '';
04  if Not(Trim(pScript) <> '') then
05   Exit;
06
07  Result := 'Script Executado: {'+#13#10 + SQL + '}';
08  QryScript.SQL.Text := pScript;
09  QryScript.ExecSQL;
10
11 end;
```

Listagem 21. Método AlterTable

```
01 function TfrmComparaDB.AlterTable(pTabela: TTabela): String;
02 Var SQL : String;
03 begin
04  Result := '';
05
06  SQL := '';
07
08  SQL := SQL + #13#10 + VerificarAlteracoesEmColunas(pTabela);
09  SQL := SQL + #13#10 + VerificarAlteracoesEmIndices(pTabela);
10  Result := ExecutarScript(SQL);
11
12 end;
```

Essa é uma função bastante simples que delega as tarefas às demais. Ela é responsável por agrupar os scripts gerados para colunas e índices e passa-los para o *ExecutarScript* que acabamos de ver, concatenando o resultado desse método à sua variável *Result*. O método *VerificarAlteracoesEmColunas* utilizado na linha 8 pode ser visto na **Listagem 22**.

Esse método inicialmente executa uma consulta na query de colunas e em seguida percorre a lista equivalente no dicionário, analisando se a coluna foi retornada na consulta ao banco. Se não existir ela deve ser criada, responsabilidade que é passada ao método *AddColuna* (**Listagem 23**), e caso essa coluna já exista é repassada a responsabilidade para o método *AlterColuna* (**Listagem 24**).

Nesse código foi adicionada uma série de *ifs* para analisar e montar a instrução *ADD COLUMN*, retornando a parte do script com as informações da coluna.

Aqui criamos uma variável chamada *Diferenca* e verificamos se existe alguma divergência entre a estrutura atual do banco e o dicionário. Caso exista, montamos o trecho de código responsável pela alteração da coluna no banco de dados, finalizando assim a criação e alteração das colunas.

Veremos agora na **Listagem 25** o método *VerificaAlteracoesEmIndices*, invocado na linha 9 do método *AlterTable* da **Listagem 21.**

Esse método é muito semelhante ao anterior. Inicialmente é feita a consulta na tabela de índices com os devidos parâmetros (linhas 8 a 11), em seguida percorremos a lista de índices cadastrados no

Listagem 22. Método VerificarAlteracoesEmColunas

```
01 function TfrmComparaDB.VerificarAlteracoesEmColunas(pTabela: TTabela) :
    String;
02  Var
03   Coluna   : TColuna;
04   Command  : String;
05   sTabela  : String;
06   i        : Integer;
07  begin
08   QryColunas.Close;
09   QryColunas.ParamByName('TABLE_SCHEMA').AsString :=
     QryTabelas.FieldByName('TABLE_SCHEMA').AsString;
10   QryColunas.ParamByName('TABLE_NAME' ).AsString :=
     QryTabelas.FieldByName('TABLE_NAME').AsString;
11   QryColunas.Open();
12
13   Result := '';
14   sTabela := edt_Schema.Text+'.'+pTabela.Nome;
15   for i := 0 to pTabela.Colunas.Count -1 do
16   Begin
17    Coluna := pTabela.Colunas.Items[i];
18    mmoLog.Lines.Add('Comparando Coluna -> ' + Coluna.Nome);
19    if Not(QryColunas.Locate('COLUMN_NAME',
     Coluna.Nome,[loCaseInsensitive])) then
20     begin
21      Result := Result + ' ' +
22            'Alter Table ' + sTabela  +  AddColuna(Coluna) + ';'+#13#10;
23     end
24    else
25     begin
26      Command := Trim(AlterColuna(Coluna));
27      if (Command <> '') then
28       Result := Result + ' ' +
29            'Alter Table '+ sTabela +  ' CHANGE COLUMN ' + Coluna.Nome +
30            ' ' + Coluna.Nome + ' ' + Command + ';'+#13#10;
31     end;
32   End;
33  end;
```

Listagem 23. Método AddColuna

```
01 function TfrmComparaDB.AddColuna(pColuna: TColuna): String;
02 begin
03  Result := ' ADD COLUMN' + ' ' +
04        pColuna.Nome + ' ' +
05        pColuna.Datatype + '(' + pColuna.Size.ToString+'' +
06        iif(pColuna.Decimais > 0,
07          ','+pColuna.Decimais.ToString,'') +')' + ' ' +
08        iif(pColuna.Not_Null,'Not null','') + ' ' +
09        iif(pColuna.Auto_Increment,'auto_increment','')
10 end;
```

dicionário e por fim testamos para checar se o índice precisa ser criado ou alterado (linha 19). Se o índice não existir no banco de dados é chamando o método *CreateIndice* (**Listagem 26**) e caso exista é utilizado o método *AlterIndex* (**Listagem 27**), verificando se ele retornou alguma coisa. Se sim é criado o script responsável pela alteração do índice (linha 28).

Esse código é bem simples e objetivo: apenas verificamos se o tipo de chave é primaria ou não para criar o script correto.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 24. Método AlterColuna

```
01  function TfrmComparaDB.AlterColuna(pColuna: TColuna): String;
02  Var Diferenca : String;
03  begin
04   Diferenca := '';
05   if(pColuna.Not_Null.ToInteger <>
06     QryColunas.FieldByName('NOT_NULL').AsInteger) Then
07    Diferenca := 'OBRIGATORIO-';
08
09   if(pColuna.auto_increment.ToInteger <>
10     QryColunas.FieldByName('AUTO_INCREMENT').AsInteger) Then
11    Diferenca := 'AUTO_INCREMENT-';
12
13   if (Trim(Diferenca) <> '') or
14     (QryColunas.FieldByName('DATA_TYPE').AsString.ToUpper  <>
15        pColuna.Datatype.ToUpper) or
16     (QryColunas.FieldByName('SIZE').AsInteger <> pColuna.Size) or
17     (QryColunas.FieldByName('NUMERIC_SCALE').AsInteger <>
        pColuna.Decimais) then
18   Begin
19    Result := pColuna.Datatype + '('+ pColuna.Size.ToString +
20           iif(pColuna.Decimais > 0,
21           ','+pColuna.Decimais.ToString,'')+')'+ ' ' +
22           iif(pColuna.auto_increment,'AUTO_INCREMENT','')+ ' ' +
23           iif(pColuna.Not_Null,'NOT NULL', NULL ) + ' ';
24   End;
25  end;
```

Listagem 25. Método VerificarAlteracoesEmIndices

```
01 function TfrmComparaDB.VerificarAlteracoesEmIndices(pTabela: TTabela):
    String;
02  Var
03    Indice   : TItemIndice;
04    Command  : String;
05    sTabela  : String;
06    i      : Integer;
07  begin
08    QryIndices.Close;
09    QryIndices.ParamByName('TABLE_SCHEMA').AsString :=
      QryTabelas.FieldByName('TABLE_SCHEMA').AsString;
10    QryIndices.ParamByName('TABLE_NAME' ).AsString :=
      QryTabelas.FieldByName('TABLE_NAME').AsString;
11    QryIndices.Open();
12
13    Result := '';
14    sTabela := edt_Schema.Text+'.'+pTabela.Nome;
15    for i := 0 to pTabela.Indices.Count -1 do
16    Begin
17      Indice := pTabela.Indices.Items[i];
18      mmoLog.Lines.Add('Comparando Indices -> ' + Indice.Nome);
19      if Not(QryIndices.Locate('INDEX_NAME',
        Indice.Nome,[loCaseInsensitive])) then
20        begin
21          Result := Result + ' Alter Table ' + sTabela + ' ' +
22                CreateIndice(Indice) +#13#10;
23        end
24      else
25        begin
26          Command := Trim(AlterIndice(Indice));
27          if (Command <> '') then
28            Result := Result + ' Alter Table '+ sTabela + ' ' +  Command +#13#10;
29        end;
30    End;
31  end;
```

Listagem 26. Método CreateIndice

```
01 function TfrmComparaDB.CreateIndice(pIndice: TItemIndice): String;
02 begin
03   Result := 'ADD ' + iif(UpperCase(pIndice.Nome) = 'PRIMARY',
04                  'PRIMARY KEY','INDEX') + ' ' +
05          pIndice.Nome + '(' + pIndice.Campos + ');'
06 end;
```

Listagem 27. Método AlterIndice

```
01 function TfrmComparaDB.AlterIndice(pIndice: TItemIndice): String;
02 begin
03   if (QryIndices.FieldByName('COLUMN_NAME').AsString <> pIndice.Campos)
04 then
05   Begin
06     if (UpperCase(pIndice.Nome) = 'PRIMARY') Then
07       Begin
08         Result := 'DROP PRIMARY KEY;' + #13#10+
09                'ADD  PRIMARY KEY'  + '('+ pIndice.Campos +');';
10       End
11     else
12       Begin
13         Result := 'DROP INDEX '+ pIndice.Nome  +';'+ #13#10 +
14                'ADD  INDEX '+ pIndice.Nome  +
15                '('+ pIndice.Campos +');';
16       End
17   End;
18 end;
```

O AlterIndice, por sua vez, não é tão simples quanto o *CreateIndice*. Nele temos uma validação para identificar se o tipo de índice é primary ou não (linha 6) para que possamos criar o script de maneira correta. Antes de criar o script de alteração devemos efetuar um *drop* e em seguida um *add*, uma vez que para índices não existe o comando *alter*, então devemos remover e recriar o índice.

Para uma melhor visualizar a interação dos métodos podemos visualizar esse contexto na **Figura 8**, através da qual podemos analisar o fluxo do código com os possíveis caminhos a serem seguidos para alterar ou criar os elementos do banco de dados.

Com a implementação concluída, basta que testemos a aplicação. Para isso só precisamos alterar o arquivo *dicionario.xml* da forma que desejarmos e informar o schema com o qual trabalharemos.

Utilizando o dicionário na aplicação

Para a última etapa nesse projeto vamos fazer um pequeno uso do dicionário na nossa aplicação para validar sua aplicabilidade. Para isso vamos criar um formulário e nele vamos efetuar duas implementações, uma para carregar os captions dos campos a partir do dicionário e apresentar com um asterisco se o campo é obrigatório, e a outra para validar se o campo está preenchido no botão *Salvar*, assim não permitindo enviar para o banco campos *not null* em branco. Para começar vamos criar um formulário como o da **Figura 9**.

Nesse form usamos componentes do tipo *TLabeledEdit* cujos labels receberam os nomes dos campos com a palavra "Campo" na frente. Com o formulário ponto devemos fazer algumas implementações no código, que podem ser vistas na **Listagem 28**.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

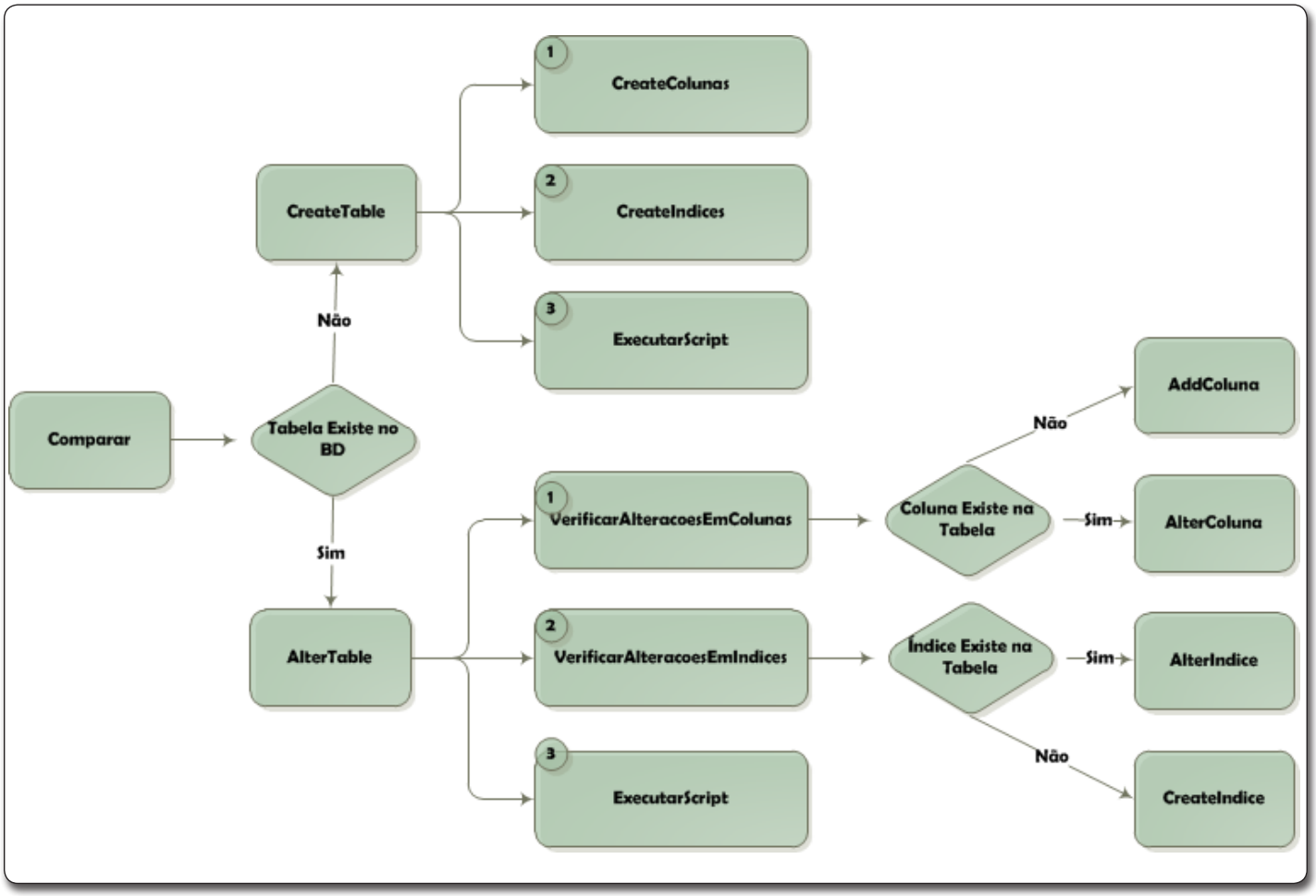


Figura 8. Fluxo dos métodos de atualização do banco de dados

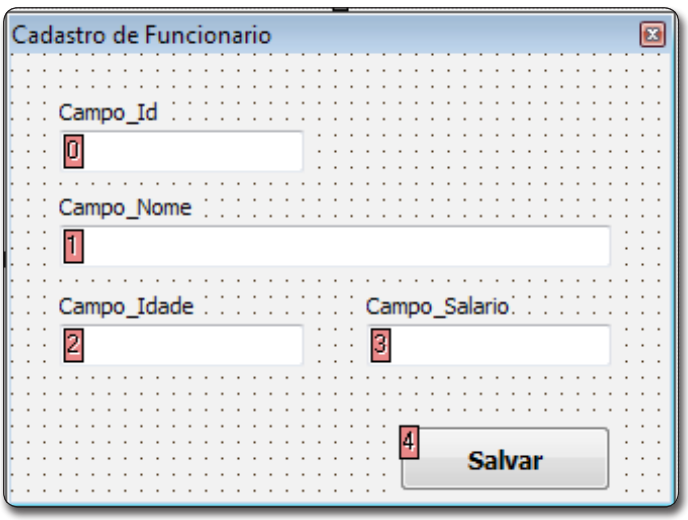


Figura 9. Formulário de cadastro de funcionário

Nota

Aqui utilizaremos como exemplo uma tabela de funcionários cuja estrutura foi devidamente definida no arquivo dicionário.xml.

Listagem 28. Classe e função a serem criadas no form

```
01 TImplementacao = Function (pColuna : TColuna;pCampo : TLabeledEdit) :
 02 Boolean of Object;
03
04 TRelacaoCampoComponente = Class
05   public
06     Campo          : String;
07     Componente : TLabeledEdit;
08 End;
```

Na unit do form vamos implementar duas classes. A *TImplementacao* servirá para implementar um "*template method*" a nível de procedure, evitando que precisemos repetir código. A *TRelacaoCampoComponente* é para que possamos criar uma lista com a relação de que coluna do nosso dicionário é representada por qual campo no formulário. Depois disso, na seção *private* da classe do formulário devemos adicionar um field que será nossa lista de campos. Para isso implemente o seguinte código:

```
FListaCampos : TObjectList;
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Vamos agora implementar o construtor do formulário, cujo código está contido na **Listagem 29**.

Primeiramente instanciamos o objeto *FListaCampos* (linha 4) e em seguida executamos um método chamado *Relacionar* passando dois parâmetros para ele, sendo uma string e um componente TLabeledEdit. Esses parâmetros são o nome da coluna no dicionário e o campo que o representará no formulário. Veja que relacionamos todos os campos da tabela de funcionário definida no XML com os campos da tela. O código desse método encontra-se na **Listagem 30**.

Listagem 29. OnCreate do formulário de cadastro de funcionários

```
01 procedure TForm1.FormCreate(Sender: TObject);
02 Var oRxC : TRelacaoCampoComponente;
03 begin
04   FListaCampos := TObjectList.Create;
05
06   //Relacionar os campos os componentes
07   Relacionar('id_Tabela1',lbledt_Id      );
08   Relacionar('Nome'    ,lbledtEdt_Nome  );
09   Relacionar('idade'   ,lbledtEdt_Idade );
10   Relacionar('Salario' ,lbledtEdt_Salario);
11 end;
```

Listagem 30. Método Relacionar

```
01 procedure TForm1.Relacionar(pCampo: String; pComp: TLabeledEdit);
02 Var oRxC : TRelacaoCampoComponente;
03 begin
04   oRxC                  := TRelacaoCampoComponente.Create;
05   oRxC.Campo            := pCampo;
06   oRxC.Componente := pComp;
07   FListaCampos.Add(oRxC);
08 end;
```

Esse método apenas cria um objeto do tipo *TRelacaoCampoComponente* e passa para as propriedades *Campo* e *Componente* desse objeto os valores recebidos por parâmetro.

Para melhor reaproveitar o código vamos criar um método que será um template responsável por percorrer a lista de campos relacionados e localizar seu equivalente no dicionário, como vemos na **Listagem 31**.

Esse método recebe como parâmetro a tabela que ele deve utilizar do dicionário, a lista de relacionamentos com a qual ele irá trabalhar e o método que será executado para cada componente que estiver relacionado com uma coluna no dicionário. A variável *oTabela* recebe uma instância da tabela retornada pela função *Retornar* a partir do nome da tabela recebida por parâmetro. Caso a tabela não exista irá ocorrer um erro, pois não foi feito um tratamento para verificar se a tabela foi realmente encontrada, então o leitor pode adicionar essa validação, como foi feito para a coluna na linha 22. Depois de passarmos um objeto para o *oTabela* é feito um laço sobre a lista de relacionamentos, onde é verificado se aquele campo especificado na lista existe no dicionário. Caso exista, executamos o método que foi passado como parâmetro para que ele faça a validação ou implementação necessária.

Para trabalhar com o template vamos criar dois métodos de "implementação", o *ValidarCampo* e o *CarregaCaption*, cujas implementações encontram-se nas **Listagens 32** e **33**, respectivamente.

Listagem 31. Método AcaoSobreDicDados

```
01 function TForm1.AcaoSobreDicDados(pTabela: String; pCampos: TList;
02   pImplementacao: TImplementacao): Boolean;
03 Var
04   DicDados : TDicDados;
05   i              : Integer  ;
06   oTabela    : TTabela  ;
07   oColuna    : TColuna  ;
08   sColuna    : String   ;
09   oEdt         : TLabeledEdit;
10 begin
11   DicDados := TDicDados.Create;
12   try
13    DicDados.CarregarDicionario;
14
15    oTabela := DicDados.Tabelas.Retornar(pTabela);
16    for I := 0 to pCampos.Count-1 do
17    Begin
18     sColuna := TRelacaoCampoComponente(pCampos[i]).Campo;
19     oEdt      := TRelacaoCampoComponente(pCampos[i]).Componente;
20     oColuna := oTabela.Colunas.Retornar(sColuna);
21
22     if Not(Assigned(oColuna)) then
23       Continue;
24
25     Result := pImplementacao(oColuna,oEdt);
26    End;
27   finally
28    FreeAndNil(DicDados);
29   end;
30 end;
```

Listagem 32. Método ValidarCampo

```
01 function TForm1.ValidarCampo(pColuna : TColuna;pCampo : TLabeledEdit) : 02
Boolean;
03 begin
04   Result := True;
05   if (pColuna.Not_Null and  (Trim(pCampo.Text) = '')) then
06   Begin
07    ShowMessage('Campo '+ pColuna.Caption + ' obrigatorio!');
08    pCampo.SetFocus;
09    Result := False;
10    Abort;
11   End;
12 end;
```

O método *ValidarCampo* recebe por parâmetro uma coluna e um *TLabelEdit* que é utilizado para validar se o campo é obrigatório e se está preenchido. Na linha 5 é utilizando o *pColuna* do nosso dicionário para saber se ele foi definido como obrigatório e na sequência, se é obrigatório, validamos se o campo está vazio. Caso sim, retornamos uma mensagem informando que o campo que está com problema, colocamos o foco nele, atribuímos *false* ao *Result* e cancelamos os próximos procedimentos.

O método *CarregarCaption*, assim como o anterior, recebe por parâmetro uma coluna e um componente. A intenção desse método é atribuir ao componente o caption definido no dicionário para o campo.

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Programador Java: Por onde começar?

Descubra nesse vídeo como entrar na carreira Java com o pé direito!

DEVMEDIA

http://www.devmedia.com.br/programador-java-por-onde-comecar/33638

Além disso estamos validando se a coluna é *not null*, se for, acrescentamos um asterisco na frente do caption para informar que esse campo é obrigatório.

Com os métodos "implementadores" definidos vamos criar outros dois que farão a chamada do método template, passando para ele a implementação a ser utilizada. Podemos criar então os métodos *ValidarCampos* e *CarregarCaptions* que podem ser vistos nas **Listagens 34** e **35**.

Listagem 33. Método CarregarCaption

```
01 function TForm1.CarregaCaption(pColuna: TColuna;
02   pCampo: TLabeledEdit): Boolean;
03 begin
04   if Not(pColuna.Not_Null) then
05     pCampo.EditLabel.Caption := pColuna.Caption
06   else
07     pCampo.EditLabel.Caption := '*' + pColuna.Caption;
08 end;
```

Listagem 34. Método ValidarCampos

```
01 function TForm1.ValidarCampos: Boolean;
02 begin
03   Result := AcaoSobreDicDados('Funcionario',FListaCampos,ValidarCampo);
04 end;
```

Listagem 35. CarregarCaptions, uso do AcaoSobreDicDados com CarregarCamptions

```
01 function TForm1.CarregarCaptions: Boolean;
02 begin
03   Result := AcaoSobreDicDados('Funcionario',FListaCampos,CarregaCaption);
04 end;
```

Esses métodos apenas fazem a chamada ao método *AcaoSobreDicDados*, passando a tabela que será utilizada, a lista de relação e o método que a ser executado pelo template.

Para concluirmos a implementação, no evento *OnShow* do formulário basta fazer a chamada ao método *CarregarCaptions* e no botão *Salvar* devemos fazer a chamada ao método *ValidarCampos*. Com esses passos executados, basta executarmos a aplicação e abrirmos o formulário, nesse momento os captions estarão como os textos definidos no dicionário.

Se trocarmos o captions no arquivo *dicionário.xml* isso irá refletir na tela quando o formulário for aberto novamente, evitando alterações nos fontes e preocupações com adequações em todo o sistema, pois bastará a partir de então que o texto seja alterado no arquivo.

No botão de *Salvar* teremos as validações feitas automaticamente de acordo com o que foi definido no dicionário, então basta que façamos as devidas ligações entre os componentes e o banco de dados, não sendo mais necessário validar todos os campos antes de gravar as informações.

Poderíamos ainda estender essa implementação para criar triggers e procedures no banco de dados, bem como excluir elementos que não devam mais ser utilizados (aqui foi feita apenas criação e alteração). Além disso, outras aplicações podem ser dadas ao dicionário, bastando utilizar os atributos definidos no XML, adicionando novos, caso necessário.

Uma vez implementada em todo o sistema, essa prática facilita e simplifica a instalação e manutenção dos sistemas, automatizando a atualização da base de dados e gerando um log de eventos sobre as operações que foram realizadas, facilitando assim a identificação de falhas que tenham ocorrido durante o processo.

Autor

Gutierry Antonio
gutierrydsn@hotmail.com

Entusiasta de Big Data e Data Warehouses, atua como Software Engineer e MySQL DBA. É graduado em Sistemas da Informação e Análise de Sistemas, e atualmente cursa especialização em Engenharia de Sistemas. Possui experiência em desenvolvimento com Delphi e Ruby on Rails com bancos de dados Firebird e MySQL, além de conhecimento de Java, C# e C++ nos bancos de dados SQL Server e Oracle, bem como de AWS.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Como corrigir Bad Smells com refatoração em Delphi

Detectando e corrigindo problemas no código fonte

Este artigo é do tipo mentoring

saiba mais: www.devmedia.com.br/mentoring-saibamais

Cenário

Este artigo discute boas e más práticas de programação em Delphi, tratando do uso de código limpo (clean code) e de problemas que dificultam a manutenção de sistemas Delphi (bad smells, ou code smells). Conforme os sistemas Delphi são evoluídos, modificados e adaptados, seu código e outros artefatos envolvidos se tornam mais complexos, se afastam de seu objetivo original e perdem qualidade. Metodologias e ferramentas não resolvem este problema, pois sua utilização normalmente auxilia na aceleração do desenvolvimento, no caso de ferramentas RAD. Em muitos projetos, boa parte do custo é dedicada à sua manutenção. Dessa forma, é necessário facilitar a manutenibilidade e legibilidade do código, melhorando sua qualidade interna. Usando boas práticas, veremos como alguns fundamentos de código limpo e refactoring podem tornar um software mais fácil de ser mantido e evoluído. Veremos como resolver problemas comuns encontrados em código fonte (bad smells), usando uma abordagem de desenvolvimento corretiva e evolutiva.

De acordo com Kent Beck, *bad smells* são estruturas no código que sugerem a possibilidade de refatoração, como nos exemplos a seguir:

- Rigidez – uma alteração em um ponto do código requer alterações em cascata (dependências entre units, por exemplo);
- Fragilidade – A modificação de um ponto do código quebra outras funcionalidades (form vs data module, por exemplo);
- Complexidade – Arquitetura muito complexa, pois foi preparada para manipular qualquer tipo de possibilidade;
- Duplicação – Código redundante, duplicado;
- Legibilidade – Código difícil de compreender.

Já o Código Limpo é uma prática que visa a escrita de código legível, funcional, pequeno, simples, fácil de entender e manter. Deve ser facilmente acessível a outros, com uma clara intenção, sem ambiguidades, boas abstrações, bons nomes para units, métodos, classes, forms, data modules, etc. Se um código está difícil de ser entendido, ele deve ser refatorado. Normalmente programadores comentam código difícil de ser compreendido.

Refatorar é reestruturar um sistema de software aplicando uma série de transformações sem modificar seu comportamento observável, afim de torná-lo mais fácil de entender e modificar. Cada pequeno passo melhora o projeto do código, mantendo seu comportamento externo observável. Existem várias motivações para o uso de refatorações, para aplicar técnicas de clean code. Primeiro, elas tornam mais fácil a adição de novos trechos de código. Quando se inclui uma nova funcionalidade em um sistema de software, existem basicamente duas opções: programar rapidamente sem se preocupar com o quanto ela se encaixa bem no projeto existente, ou pode-se modificar o projeto existente a fim de melhor acomodar essa funcionalidade. No primeiro caso, ocorre um *débito de projeto*, o qual pode ser pago mais tarde com refatoração. Segundo, as refatorações melhoraram um projeto de código existente, no momento que tornam o código mais simples, claro e mais fácil de trabalhar, manter e evoluir.

O uso de refatoração como uma abordagem simples estimulou vários esforços para desenvolver abordagens semiautomáticas para detectar falhas de projeto. A aplicação correta de refatorações apropriadas em um determinado contexto aumenta a qualidade do projeto, sem alterar o seu comportamento. No entanto, a identificação de inconsistências no código fonte não é uma tarefa simples, tais como métodos, fragmentos de métodos e atributos que devem ser movidos para outras classes.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

A motivação para melhorar o projeto de um sistema de software é encontrar problemas no código fonte e usar refatoração como uma possível solução. As limitações (*bad smells*) descrevem problemas e uma lista de refatorações relacionadas que podem ajudar a melhorar o código. A refatoração se concentra principalmente no tratamento dessas limitações, mas a implementação de melhorias depende das habilidades do desenvolvedor, que realiza manutenções de software. Encontrar limitações pode envolver inspecionar todo o código fonte, o que pode se tornar impraticável para sistemas de médio e grande porte. Neste cenário, o apoio semiautomático para detectar falhas é essencial. Felizmente, o RAD Studio oferece várias ferramentas para esse suporte.

Então por que as aplicações Delphi são as que apresentam os maiores débitos de projeto atualmente? Por que é difícil migrar para uma nova plataforma, um novo engine de acesso a dados, reaproveitar código, dar manutenção, trocar de versão do Delphi, enfim, evoluir sistemas Delphi? Vamos fazer uma análise do desenvolvimento Delphi, rebuscando a história das linguagens RAD. Antes do surgimento do Delphi 1, que foi um marco para o desenvolvimento visual na época, tínhamos apenas o VB como forte concorrente. Foi exatamente nesta época que surgiu o conceito de "desenvolvimento visual", ou as "linguagens visuais", incluindo aí o então IDE do Visual Studio a seguir. O termo "visual" se originou do fato de que as linguagens de programação não tinham um ambiente gráfico de design, basicamente você programava num editor de textos "às escuras", compilava, rodava e torcia para que os elementos gráficos ficassem organizados como previsto no código. Esse "apelo" trazido pelas linguagens visuais, como Delphi e VB, tornaram o desenvolvimento extremamente rápido. Aquela barra de menu que se levava duas semanas para construir estava ali pronta na caixa de ferramentas, basta fazer o "*drag and drop*" e pronto. Antes mesmo de compilar, você já tem o visual da interface

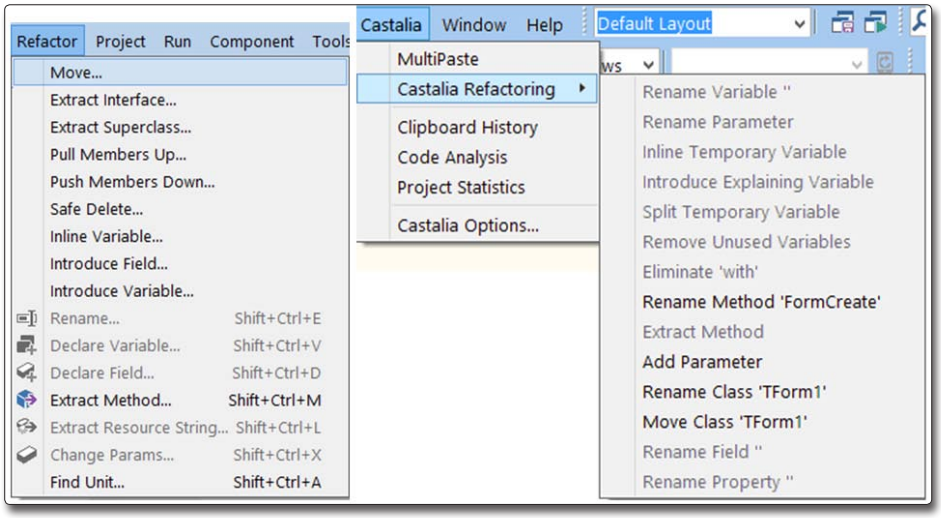


Figura 1. Menu Refactor e Castalia Refactoring

de usuário. Esse desenvolvimento rápido introduziu outro termo amplamente utilizado até hoje, o RAD – *Rapid Application Development*, o desenvolvimento rápido de aplicações. Aqui entra o fator crítico e o elo dos "studios" (RAD, Visual etc.) com a Refatoração: a programação seguindo esta abordagem RAD permite que muitos desenvolvedores abram mão de boas práticas de orientação a objetos e padrões de projeto para obter um mesmo resultado mais rapidamente, seja por falta de conhecimento, pressão devido a prazos curtos ou mesmo para reduzir custos, abrindo mão da qualidade. O sistema parece funcionar bem por fora, mas por dentro está mal projetado, ou não tem projeto, ou está um verdadeiro caos que nem mesmo o programador original o entende. Todo o código está em um único formulário ou *data module*. Não existem classes de negócio, e as poucas que existem estão fortemente acopladas. Conforme a aplicação cresce, mais componentes, código e funcionalidades (e bugs, muitos deles) são adicionados ao projeto original. Para muitas empresas, se não a maioria delas (exceto startups), o foco principal não é criar software rapidamente, mas software que seja fácil de ser mantido ao longo do tempo, pois isso reduz custos a longo prazo.

Conhecer os bad smells, patterns e anti-patterns, saber como encontrá-los e eliminá-los, usando as ferramentas oferecidas pelo RAD Studio, se torna uma tarefa vital no ciclo de vida de um sistema de software Delphi, seja para melhorá-lo, ou para evolui-lo. Dessa forma, vamos enumerar a seguir os principais mau cheiros encontrados em aplicações Delphi e como você pode usar o suporte integrado a refactoring (disponível no menu principal do IDE, como mostra a **Figura 1**) para eliminar esses problemas. Além disso, também veremos como os recursos de Métricas e Auditoria de Código auxiliam nesse processo.

Código Duplicado

Este problema consiste de duas expressões idênticas em dois métodos na mesma classe, ou em duas classes não associadas. Essa duplicação pode ser eliminada aplicando-se a refatoração *Extrair Método*. Nesse caso, o código é implementado uma única vez e chamado no local de origem onde estava o código duplicado, sendo então chamado de outros locais. Se o código duplicado encontra-se em duas classes distintas, pode-se aplicar a refatoração *Extrair Classe*, criando-se um único objeto que por delegação recebe a chamada compartilhada aos métodos originalmente duplicados. Um ótimo padrão de projeto que pode ser utilizado para eliminar a duplicação de código é o *Template Method*. Se duas subclasses possuem códigos idênticos chamados na mesma sequência, isso pode formar um método na classe base que

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

é então chamado pelas classes descendentes. Se alguma classe precisa refinar algum destes passos, então pode-se aplicar uma chamada abstrata na classe base e com polimorfismo faz-se um *override* no passo distinto na classe descendente.

Sem dúvida este é um dos piores bad smells que podem existir em aplicações. Quando um programador dá nome para suas classes, formulários, data modules, componentes, métodos, variáveis, deve fazer isso de maneira bem-feita, usando técnicas de clean code, que revelem bem a sua intenção. Sempre que possível, incluir documentação a respeito da classe/método/funcionalidade implementada no sistema. Isso porque se um programador não conseguir encontrar aquela rotina que realiza determinada tarefa no sistema, normalmente escreverá novo código, gerando retrabalho e duplicidade.

O IDE do RAD Studio oferece uma opção, no menu *Project* > *QA Audits* (**Figura 2**), para procurar por código duplicado no projeto, inclusive sugerindo algumas *refactorings* e *patterns* que podem ser aplicados. Caso encontre código duplicado, o IDE mostrará o local exato. Clicando-se sobre o item, o editor posicionará no local do código fonte.

Método Longo

Consiste de um método com uma implementação de lógica com várias, dezenas ou até centenas de linhas de código. Vários problemas podem ser originados da implementação de métodos longos. Primeiro, pode existir código duplicado. Segundo, será difícil reaproveitar parte do código em outro local.

A solução é dividir o método longo em métodos menores, usando-se a refatoração *Extrair Método*.

Normalmente, um método não deve conter mais do que 20 linhas, segundo boas práticas de clean code. Alguns autores sugerem 10 linhas e outros, no máximo cinco. Bons métodos podem conter apenas uma ou duas linhas de código, normalmente um evento de um controle de tela que dispara um método de negócio, como um padrão *Command*. Atente para o uso de variáveis locais, elas tendem a encorajar a escrita de métodos muito longos.

Não há um número bem definido de linhas de código que é considerado adequado para um método, portanto, uma boa forma de mantê-lo enxuto é garantir que ele desempenhe uma única função bem definida.

Classe Larga (ou Classe "Deus")

Uma boa prática da programação orientada a objetos sugere que uma classe deve ter uma única responsabilidade (Princípio da Responsabilidade Única – o S dos padrões SOLID). Quando uma classe possui muitas instâncias no código, isso sugere que ela está fazendo muito, ou pelo menos realizando trabalho que deveria ser feito por outras classes. Algumas refatorações podem ajudar a reduzir classes longas, como *Extrair Classe* e *Extrair Subclasse*.

No Delphi, na própria VCL, podemos observar dezenas de classes desse tipo, que realizam muitas tarefas, delegando pouco. Atente para grandes cadeias hierárquicas de classes. Normalmente, você deve ter dois ou três níveis de hierarquia (herança) em seu framework, sendo que o superior é uma abstração (classe abstrata). Isso porque grandes níveis de hierarquia vão tender a sobrecarregar as classes descendentes com muitas responsabilidades. Uma subclasse não escolhe o que herda, por padrão tudo que é *published*, *public* e *protected* é herdado. Existe uma métrica chamada *Depth of Inheritance* (DIT, ou DOI), avaliada pelo QA Metrics do IDE, que permite mensurar essa complexidade.

Outro problema com a herança é que ela é muito pública, explícita. Prefira a herança de caixa preta para adicionar funcionalidades a uma classe sem o uso de herança. Essa é uma excelente prática, pois permite esconder o objeto delegado, simulando herança múltipla, ou ainda oferecendo a possibilidade de se trocar o

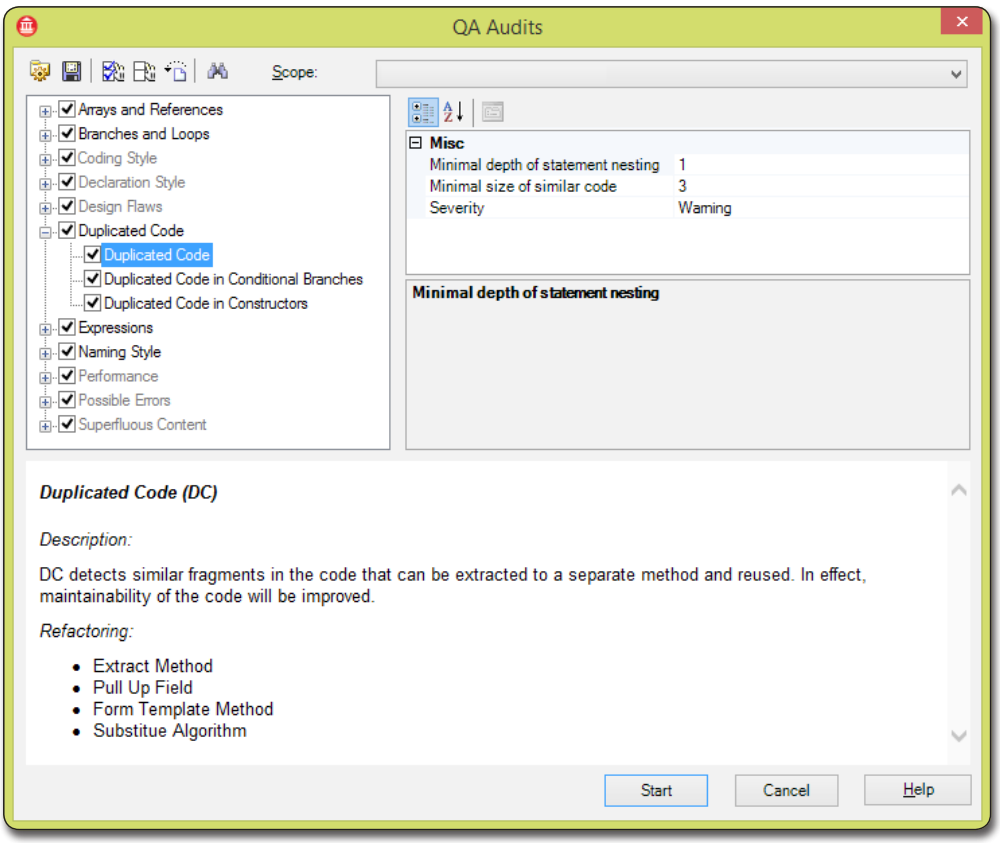


Figura 2. Auditoria para procura por código duplicado

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

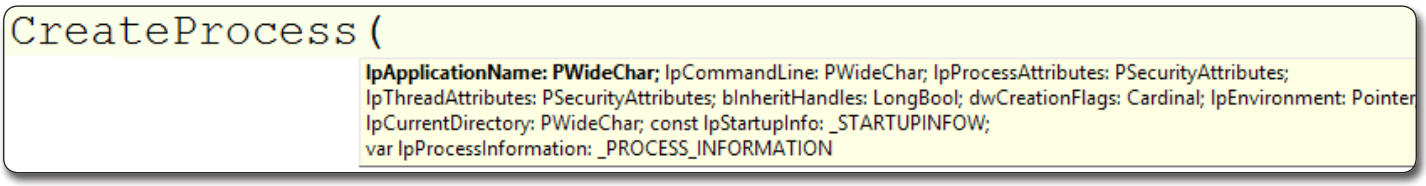


Figura 3. API CreateProcess com uma lista enorme de parâmetros

objeto interno, o que já seria muito complicado de se fazer com herança. Imagine mudar a classe base de um componente amplamente usado, por exemplo. A refatoração *Substituir Herança por Delegação* ajuda a resolver problemas desse tipo. A **Listagem 1** mostra um exemplo, recurso também conhecido como *Introduzir Método Estrangeiro*.

Listagem 1. Herança de caixa preta

```
01 TClientDataSetEx = class
02 strict private
03   FCds: TClientDataSet; // não herda, delega e esconde
04 public
05   // Método a ser "adicionado" em TClientDataSet
06   procedure AlgumaFuncaoAdicional();
07   constructor Create(ACds: TClientDataSet);
08 end;
09
10 procedure TClientDataSetEx.AlgumaFuncaoAdicional();
11 begin
12   // implementação
12 end;
14
15 constructor TClientDataSetEx.Create(ACds: TClientDataSet);
16 begin
17   self.FCds := ACds;
18 end;
```

Lista Longa de Parâmetros

Um método que recebe muitos parâmetros sugere que classes possam estar compartilhando muita informação (se tornando muito "íntimas"), ou ainda, dados globais. Se um método com uma longa lista de parâmetros está sendo chamado em vários locais do código, uma mudança em um deles pode desencadear uma cadeia muito grande de alterações em vários locais do código.

Vamos analisar uma das funções da API do Windows que mais sofre desse bad smell, a *CreateProcess* (**Figura 3**). Observe a quantidade de parâmetros necessários para poder chamar a função. Se algum parâmetro não for necessário, mesmo assim você será obrigado a passar, ou 0, uma string em branco, ou pior, um objeto *nil* ou um ponteiro nulo, o que pode ocasionar *Access Violations* difíceis de serem encontrados. Boas práticas de clean code dizem que você nunca deve passar *nil*. Além disso, nesse mesmo exemplo do *CreateProcess*, vemos outro bad smell clássico conhecido com *Obsessão Primitiva* (visto a seguir), onde somente tipos primitivos da plataforma/linguagem são usados. No caso do Windows, é um legado. A VCL brilhantemente, através de vários patterns como *Façade*, encapsulou estas APIs para torná-las orientadas a objetos. De fato, VCL, Java, .NET, são todos frameworks que criam classes/tipos/objetos em cima de uma API normalmente não orientada a objetos. Por exemplo, para você criar um formulário em Delphi, você não precisa chamar uma função do Windows passando dezenas de tipos primitivos, como ponteiros, bools, strings etc., você utiliza a classe TForm, cria uma instância dela, configura propriedades, e chama métodos. Por baixo, a VCL vai mapeando tudo para você. Esse é um típico padrão de projeto (*design pattern*), que mescla *Façade, Adapter, Wrapper* e fábricas (*factories*).

A **Listagem 2** mostra um exemplo de um método com vários bad smells. Primeiro, uma lista longa de parâmetros. Segundo, uma obsessão primitiva. E finalmente, dados agregados, que sempre aparecem juntos. Tudo foi eliminado através da refatoração *Introduzir Parâmetro Objeto*, onde transformamos cada um dos parâmetros em atributos de uma nova classe. O método agora recebe um único parâmetro. Se for adicionado um novo atributo, a assinatura do método não muda, outra excelente prática (interfaces imutáveis).

Listagem 2. Método com três bad smells

```
01 // Antes
02 procedure CadastrarCliente(
     Codigo: integer; Nome, Endereco,
     Num, Comp, CEP, DDD, Fone: string);
03 ...
04 // Depois
05 TTelefone = class
06   DDD, Fone: string;
07 end;
08
09 TCliente = class
10   Codigo: integer;
11   Nome, Endereco, Num, Comp, CEP: string;
12   Fone: TTelefone;
13 end;
14 ...
15 procedure CadastrarCliente(Cliente: TCliente);
16 ...
```

Dados Agregados

Ocorre normalmente quando dados costumam aparecer juntos em várias situações: campos em classes, em lista de parâmetros em assinaturas de métodos etc. Uma solução seria aplicar a refatoração *Introduzir Parâmetro Objeto*, como visto anteriormente. A VCL do Delphi, por exemplo, tem este bad smell no método *SetBounds* (visto a seguir), que posiciona um

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

controle em determinada coordenada, com um tamanho (altura e largura). Você obrigatoriamente terá que passar estes quatro parâmetros (ALeft, ATop, AWidth, AHeight), e eles sempre aparecem juntos (daí o nome agregados).

Outros exemplos são DDD/Fone, Endereço/Num/Complemento/CEP etc. Uma boa prática seria criar um parâmetro objeto (TBounds) e então preservar o objeto inteiro ao passá-lo como parâmetro, como mostra a **Listagem 3**.

Listagem 3. Exemplo de bad smell Dados Agregados e sua solução

```
01 //Método original
02 procedure TWinControl.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
03
04 //Classe para conter os parâmetros
05 TBounds = class
06    Left, Top, Width, Height: Integer;
07 end;
08
09 //Proposta de método refatorado
10 procedure TWinControl.SetBounds(Abounds : TBounds);
```

Obsessão Primitiva

Existem basicamente dois tipos de dados na maioria das linguagens: os primitivos e as estruturas (normalmente classes, formadas a partir da composição de outros tipos primitivos). A obsessão primitiva ocorre normalmente com desenvolvedores com pouca experiência técnica em orientação a objetos, ou ainda, em sistemas com mais de década de idade que nasceram e evoluíram a partir de projetos de linguagens estruturadas.

Infelizmente, este é um dos piores legados da linguagem Delphi, a herança da linguagem Pascal. Ou mesmo a obsessão do programador que veio de linguagens estruturadas, sem orientação a objetos. Como não existia o conceito de classe, o desenvolvedor era obrigado a compartilhar informações por parâmetro, para evitar variáveis globais. Isso causa ainda uma lista longa de parâmetros. Se existisse classe, dois ou três métodos poderiam compartilhar um *field* privado da classe, por exemplo. Ou ainda, um tipo primitivo poderia ser substituído por um mais específico. É muito comum, por exemplo, encontrarmos inteiros para representar o que deveria ser objetos TCliente, TProduto, ou ainda strings. É uma prática muito comum em projetos Delphi a existência de uma unit chamada global.pas ou funcoes.pas, onde existem centenas de métodos, fora de classes, que recebem ou inteiros e strings e retornam booleanos. Esta é a obsessão primitiva, o programador inicia a criação de uma função, e tudo que lhe vem à mente para usar como parâmetro ou retorno são strings, chars, booleanos e inteiros.

A **Listagem 4** mostra um exemplo disto, do autor que vos escreve. Trata-se de parte de um programa (apenas a sessão interface) escrito há mais de 20 anos, em Turbo Pascal, sem suporte a orientação a objetos. Note a obsessão primitiva em praticamente todos os procedures/functions. O programa, que roda em DOS, controla locadoras e farmácias e seu código fonte completo está disponível no Code Central (ver seção **Links**).

Listagem 4. Programador que só sabe usar string, boolean, char, byte e integer

```
01 function time : string;
02 function Date : string;
03 procedure status(mensagem : string);
04 procedure beep(freq,time:integer);
05 procedure most(x,y,c,f:byte;st:string);
06 function getchar : char;
07 function cor_de_fundo : byte;
08 function cor_de_frente : byte;
09 procedure menu(x1,y1,x2,y2,c1,c2 : byte;text : string);
10 procedure erro (text : string);
11 function Maiuscula(st1 : string) : string;
12 function ok : boolean;
13 procedure wri(x,y : byte; text : string);
14 procedure salva_tela (i: byte);
15 procedure volta_tela (i: byte; opcao: boolean);
16 function inttostr(int : integer) : string;
17 function strtoint(st1 : string) : integer;
18 function strtoreal(st1 : string) : real;
19 function realtostr(r : real) : string;
20 procedure item(x,y : byte; text : string);
21 procedure execute (comando : string);
22 procedure cores(cor : byte);
23 procedure QuickReport (arquivo,titulo : string);
24 procedure RadioGroup (x1,y1,cor1,cor2,item : byte;var itemindex : byte);
25 procedure CheckBox (x1,y1,item : byte; var marc : string);
26 procedure square (x1,y2,cor1,cor2: byte;st : string);
27 procedure SetHint(LongHint : string);
28 procedure imprima(frase : string);
29 procedure preencha(c1,c2 : byte; st : char);
```

Sentenças Condicionais

Testar várias possibilidades com sentenças *case* tende a duplicar código. Se um novo teste for necessário na sentença, para comportar uma nova possibilidade, e esse teste for feito em vários locais, então várias alterações serão necessárias. Muitas vezes os testes podem ser substituídos por polimorfismo.

Este é um dos bad smells que aparecem com mais frequência em sistemas Delphi. Sentenças condicionais são ruins por vários motivos. Primeiro, elas tornam complexa a interpretação do caminho de execução por parte do programador, que mentalmente precisa montar uma "tabela booleana" para descobrir o que vai ser executado de acordo com a condição. Segundo, condicionais que combinam vários testes, com *ANDs* e *ORs*, tornam ainda mais difícil a interpretação. Usar *cases* para testar possibilidades, sejam números mágicos (if flag = 1?), tipos enumerados ou classes, tende a duplicar código. Se um novo tipo for testado, todos os locais que fazem o teste precisam comportar o novo tipo. Dificultam ainda a cobertura por testes unitários. Pior, usar vários IFs / ELSEs encadeados, cria uma estrutura lógica complexa de interpretar, entender, com muita identação e esteticamente feia. E finalmente, um método com muitos testes fatalmente será longo.

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Felizmente, o IDE do RAD Studio possui várias ferramentas para ajudar a detectar onde estão as piores instruções condicionais do seu projeto. Você pode fazer isso através de *QA Metrics* (**Figura 4**), através da métrica Complexidade Ciclomática (**BOX 1**), ou ainda através do próprio menu *Castalia* > *Code Analysis*.

BOX 1. Complexidade Ciclomática

A complexidade ciclomática (CC) é uma métrica de software utilizada na avaliação da complexidade de um código. Essa métrica avalia a quantidade de caminhos possíveis em um código, levando em consideração estruturas de desvio de fluxo. Para cada passo no código, é somado um valor ao índice da CC, que quanto maior, indica maior complexidade no código, o que implica em maior possibilidade de erros e dificuldade de manutenção.

Quanto maior for o nível dessa métrica, mais "tóxico" estará o código.

A **Figura 5** mostra a análise de código feita pelo Castalia na unit Vcl.Forms do Delphi. Observe que o método *CreateParams* da classe *TCustomForm* está com um alto índice de Complexidade Ciclomática, e por consequência de Toxidade (manutenção). Abrindo o código fonte do método (**Listagem 5**), vemos uma estrutura condicional extremamente complexa, com vários IFs encadeados, CASEs encadeados, IFs THEN ELSEs usando expressões AND, NOT, OR etc. Fica praticamente impossível descobrir quando um trecho de código será executado segundo os valores booleanos testados. Por motivos de espaço, veremos aqui apenas parte do método, mas já dá para ter uma ideia da complexidade.

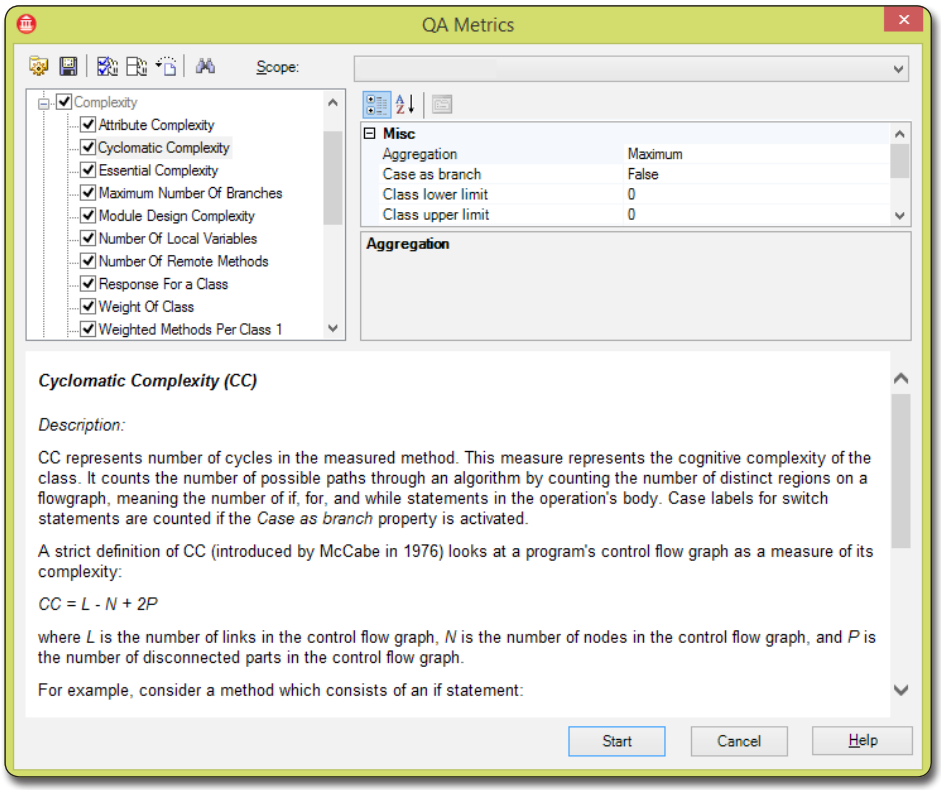


Figura 4. Complexidade Ciclomática pode ser medida pelo RAD Studio

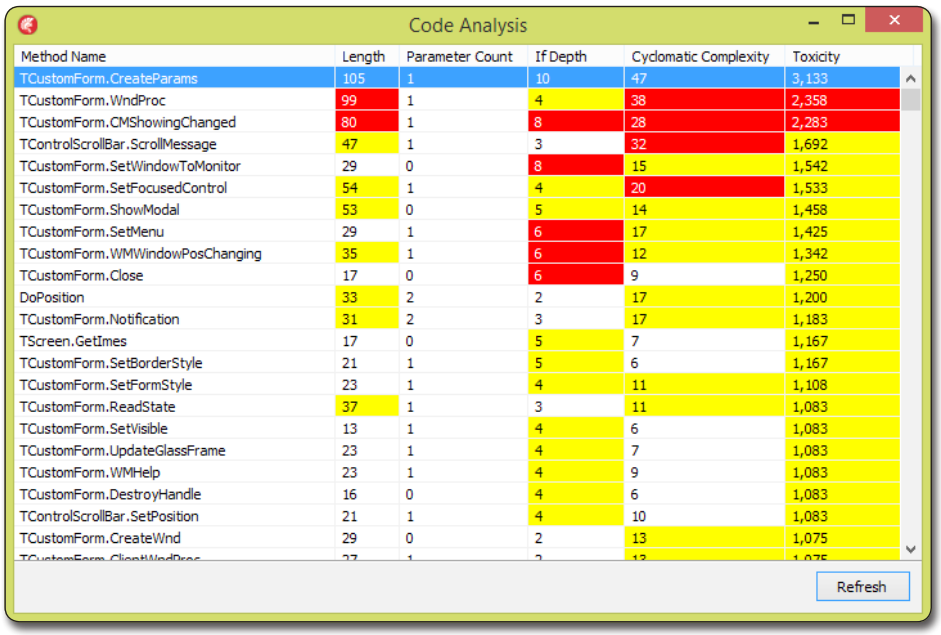


Figura 5. Método CreateParams de TCustomForm tem CC elevada

Generalidade especulativa

Este problema ocorre quando se projeta uma solução genérica para uma possível situação que possivelmente nunca ocorrerá, tornando objetos flexíveis o suficiente para tratar qualquer tipo de situação, por exemplo, criando-se classes abstratas que não realizam muito. Outro ponto é o uso de variáveis primitivas para poderem receber qualquer coisa, como ponteiros, inteiros, handles, ou pior, variants ou TObjects. Quem recebe esses objetos não tem a mínima ideia do que são, de que métodos podem ser chamados, normalmente terão que recorrer a RTTI/reflexão para poder obter algo, fazer *type casts*, ou correr o risco de acessar algo que não existe ou de um tipo inválido.

Vamos a um exemplo clássico da VCL. TList é uma classe que permite armazenar uma coleção de objetos (TObject). Se você abrir o código fonte da VCL, verá que essa coleção não só permite armazenar qualquer objeto (botões, forms, datasets),

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 5. Método com alto índice de complexidade ciclomática

```
01 procedure TCustomForm.CreateParams(var Params: TCreateParams);
02 var
03   LRect: TRect;
04   LParent: TCustomForm;
05   CreateStyle: TFormBorderStyle;
06   LPopupMode: TPopupMode;
07 begin
08   inherited CreateParams(Params);
09   InitAlphaBlending(Params);
10   with Params do
11   begin
12    if (Parent = nil) and (ParentWindow = 0) then
13    begin
14     LParent := nil;
15     if csDesigning in ComponentState then
16      LPopupMode := pmExplicit
17     else if (fsModal in FormState) and (FPopupMode = pmNone) then
18      LPopupMode := pmAuto
19     else if FormStyle = fsNormal then
20      LPopupMode := FPopupMode
21     else
22      LPopupMode := pmNone;
23     if (FInternalPopupParent = nil) and (FInternalPopupParentWnd = 0) then
24      case LPopupMode of
25       pmNone:
26        begin
27         if Application.MainFormOnTaskBar then
28         begin
29          // FCreatingMainForm is True when the MainForm is
30          // being created, Self = Application.MainForm during CM_RECREATEWND.
31          if FCreatingMainForm or (Self = Application.MainForm) then
32           WndParent := 0
33          else
34           if Assigned(Application.MainForm) and Application.MainForm
             .HandleAllocated then
35           begin
36            WndParent := Application.MainFormHandle;
37            if WndParent = Application.MainForm.Handle then
38            begin
39             if Application.MainForm.PopupChildren.IndexOf(Self) < 0 then
40              Application.MainForm.PopupChildren.Add(Self);
41             FreeNotification(Application.MainForm);
42            end;
43           end
44           else
45            WndParent := Application.Handle;
46         end
47         else
48         begin
49          WndParent := Application.Handle;
50          SetWindowLong(WndParent, GWL_EXSTYLE, GetWindowLong
             (WndParent, GWL_EXSTYLE) and not WS_EX_TOOLWINDOW);
51         end;
52        end;
53       pmAuto:
54        begin
55         if FCreatingMainForm then
56          WndParent := 0 // A main form can't be parented to another form
57         else
58          WndParent := Application.ActiveFormHandle;
59         if (WndParent <> 0) and (IsIconic(WndParent) or not
           IsWindowVisible(WndParent) or
60          not IsWindowEnabled(WndParent)) then
61          WndParent := 0;
62         if (WndParent <> 0) and
63          (GetWindowLong(WndParent, GWL_EXSTYLE)
            and WS_EX_TOOLWINDOW = WS_EX_TOOLWINDOW) then
64         WndParent := GetNonToolWindowPopupParent(WndParent);
65         if (WndParent <> 0) and (Screen.ActiveForm <> nil) and
66          (Screen.ActiveForm.WindowHandle = WndParent) then
67          LParent := Screen.ActiveForm
68         else if WndParent = 0 then
69          if Application.MainFormOnTaskBar then
70 ... // mais dezenas de linhas aqui
```

como também qualquer coisa que seja um Pointer. E no Windows, Pointer é qualquer coisa, pode ser um arquivo, um handle, o botão *Iniciar*. Isso porque quem projetou a coleção TList, teve que criar um algoritmo genérico, para que você pudesse criar coleções dos mais variados tipos de objetos, procurar por elementos, adicionar, remover, sem replicar o código desse algoritmo para cada um dos tipos de objetos existentes (o que seria impraticável). Como criar uma coleção que possa armazenar tanto TButtons, TEdits, TClientDataSets? Fácil, coloque o tipo como algo o mais genérico possível, como Pointer, ou TObject. O problema disso é que, primeiro, você poderá armazenar diferentes tipos de objetos na mesma coleção, o que muitas vezes não faz sentido. Outro problema é que ao retirar um objeto da coleção, você normalmente receberá um ponteiro, ou um TObject, e terá que "adivinhar" o que ele é, seja por RTTI ou *type cast*, o que também consome recursos. E testar tipos não é uma boa solução, nunca.

Listagem 6. Trabalhando com Generics

```
01 // Delphi 7
02 TComponent = class(
03 TPersistent, IInterface, IInterfaceComponentReference)
04 private
05   FOwner: TComponent;
06   FName: TComponentName;
07   FTag: Longint;
08   FComponents: TList;
09 // Delphi XE8
10 TComponent = class(
11   TPersistent, IInterface, IInterfaceComponentReference)
12 private
13   [Weak] FOwner: TComponent;
14   FName: TComponentName;
15   FTag: NativeInt;
16   FComponents: TList<TComponent>;
```

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Felizmente, o Delphi atualmente possui um recurso já presente em outras linguagens como Java, C# e C++: *Generics*. Com esse recurso você pode reaproveitar um algoritmo inteiro variando nele um *tipo* <T> ao invés de um objeto/variável. Observe na **Listagem 6** o código da classe TComponent do Delphi 7, e no final, a mesma classe no XE8. Note que a coleção interna de componentes de um componente (mecanismo de owner) é um TList (de pointers). Já no XE8, essa lista é um TList genérico de TComponent.

Campos Temporários / Variáveis Locais

Muitas vezes um objeto tem sua variável de instância configurada somente sobre certas circunstâncias. Tal código é difícil de entender. Um exemplo ocorre quando um algoritmo depende de uma série de variáveis. Variáveis locais em Delphi/Pascal são usadas por décadas. Além disso, uma particularidade dessas linguagens é ter uma sessão "var" local específica para declarar esse tipo de campo. Qual o problema das variáveis locais? Muitos. Primeiro, imagine se um método já sai declarando uma dezena de variáveis locais, no mínimo quantas linhas de código conterá? Centenas, o que já indica um método longo. Essas variáveis normalmente serão passadas para fora do método, ou pior, da classe, violando o encapsulamento e diminuindo a coesão.

Mais uma vez, variáveis temporárias são herança do Pascal e linguagens estruturadas, onde não havia outro local onde você pudesse declará-las (a não ser globalmente na unit). Mas existe uma forma de fugir das variáveis locais? Sim, você pode usar a técnica *Substituir Temporário por Consulta*, ou ainda, subir a variável local para o escopo privado da classe, para que seja visível por outros métodos. A **Listagem 7** mostra um exemplo de dois métodos que usavam uma variável local para consultar o valor de um Edit que contém um valor monetário a ser depositado ou sacado de uma conta corrente. A seguir, na **Listagem 8** ambos foram reestruturados para usar uma function que retorna a "variável local", agora um membro do form.

Biblioteca de Classes Incompleta

Objetos são o princípio básico da reutilização de código. Bibliotecas de classes realizam boa parte de tarefas comuns usando esse conceito. Porém, normalmente não se pode alterar uma biblioteca base de classes. A técnica de *Introduzir Método Estrangeiro*, vista anteriormente, permite simular a inclusão de um método em uma classe que não pode ser estendida nem modificada.

Imagine por exemplo que você precisa exportar um ClientDataSet para o formato TXT. Todos sabemos que é possível exportar um ClientDataSet para XML usando:

```
ClientDataSet1.SaveToFile('dados.xml');
```

Infelizmente, apesar de possível, não podemos abrir a unit do ClientDataSet e implementar o novo método. Quando um programador precisa adicionar uma nova funcionalidade em uma classe incompleta, existem várias soluções. Uma delas seria como mostrado na **Listagem 9**.

Listagem 7. Variáveis temporárias

```
01 procedure TFormBanco.btnDepositarClick(Sender: TObject);
02 var
03  Valor: double;
04 begin
05  Valor := StrToFloat(EdtValor.Text);
06  ContaCorrente.Depositar(Valor);
07 end;
08
09 procedure TFormBanco.btnSacarClick(Sender: TObject);
10 var
11  Valor: double;
12 begin
13  Valor := StrToFloat(EdtValor.Text);
14  ContaCorrente.Sacar(Valor);
15 end;
```

Listagem 8. Substituindo variáveis locais por métodos de consulta (query)

```
01 function TFormBanco.Valor(): double;
02 begin
03  result := StrToFloat(EdtValor.Text);
04 end;
05
06 procedure TFormBanco.btnDepositarClick(Sender: TObject);
07 begin
08  ContaCorrente.Depositar(Valor);
09 end;
10
11 procedure TFormBanco.btnSacarClick(Sender: TObject);
12 begin
13  ContaCorrente.Sacar(Valor);
14 end;
```

O problema aqui é que estamos usando uma abordagem clássica, estilo Pascal, sem orientação a objetos, como faz a RTL.

Então, você poderia usar a rotina dessa forma:

```
ExportToTxt(ClientDataSet1);
```

Sendo que em uma linguagem orientada a objetos, deveria ser exatamente o contrário, ou seja, você não manda exportar o ClientDataSet para TXT, você pede a ele que o faça, chamando um de seus métodos, invertendo a chamada, algo como:

```
ClientDataSet1.ExportToTxt();
```

Uma outra técnica seria criar uma classe descendente de TClientDataSet, e incorporar a nova funcionalidade. Isso seria um erro gravíssimo, pois você estaria colocando mais uma classe no nível de hierarquia simplesmente para fazer uma exportação, o que não vale o preço. Em outras palavras, seria uma *Lazy Class* (Classe Preguiçosa, o contrário da "Classe Deus"). Uma boa solução, como apresentado anteriormente, é esconder a herança e introduzir o método estrangeiro.

Porém, graças a um excelente recurso da Delphi Language, chamado de *Class Helper*, podemos simular o mesmo efeito, "injetando" um método estrangeiro em uma determinada classe do

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

framework. O código da **Listagem 10** mostra como fazer isso. Note o uso da sintaxe "*helper for*".

A partir daí qualquer lugar da sua aplicação que contiver um ClientDataSet vai poder chamar o método diretamente pelo próprio objeto, como se o mesmo tivesse sido originalmente escrito nele (**Figura 6**).

Listagem 9. Unit para incluir novas funcionalidades

```
01 unit uExport;
02
03 interface
04
05 uses
06  Datasnap.DBClient;
07
08 procedure ExportToTxt(CDS: TClientDataSet);
09
10 implementation
11
12 procedure ExportToTxt(CDS: TClientDataSet);
13 begin
14  // implementação
15 end;
16
17 end.
```

Listagem 10. Class Helper

```
01 unit uExport;
02
03 interface
04
05 uses
06  Datasnap.DBClient;
07
08 type
09  TClientDataSetEx = class helper for TClientDataSet
10    procedure ExportToTxt();
11  end;
12
13 implementation
14
15 procedure TClientDataSetEx.ExportToTxt();
16 begin
17  // aqui você tem acesso a todos os membros de TClientDataSet
18 end;
19
20 end.
```

Você pode usar o recurso para injetar métodos em qualquer classe da VCL/FMX, desde Edits, Labels, até mesmo tipos primitivos como strings e inteiros.

Comentários

Comentários são bons, porém, um código com muitos comentários é um forte indício que ele está difícil de entender. Talvez seja mais apropriado dar nomes mais sugestivos a métodos, deixando mais claras suas responsabilidades, usando a refatoração *Renomear*. Muitos programadores usam comentários como "desodorante", ou seja, para esconder o mau cheiro no código. Se um método está difícil de ser entendido, ele deve ser refatorado, usando as técnicas de clean code. Por exemplo, dar nomes pronunciáveis, sem abreviaturas (não poupe chars). Procure usar "*Camel Case*" ao dar nomes para métodos. Outro problema é que comentários não recebem manutenção como código. Programadores normalmente não mexem nos comentários, mesmo modificando o código ou seu propósito. Comentários ficam obsoletos com o tempo e podem causar confusão. O código deve ser como uma redação, o que vai dispensar o uso de comentário na maioria das vezes. Veja um exemplo na **Listagem 11**.

Para eliminar bad smells precisamos reestruturar aplicações. Refatorar é mudar o comportamento interno do sistema de software sem mudar seu comportamento externo observável. Clean code pode ajudar a tornar o código mais legível para ser entendido por outros programadores, facilitando a manutenção corretiva futura. Pode ajudar a melhorar sua estrutura interna a fim de que novos requisitos possam ser acomodados de forma mais adequada, sem introduzir bugs. A refatoração deve ser um hábito contínuo durante o desenvolvimento do software, para que não ocorra o *débito de projeto*. A equipe gasta mais tempo aplicando padrões e boas práticas, porém o software estará sempre limpo, bem estruturado, fácil de entender, manter e evoluir. Programadores devem codificar para hoje, pois não são videntes. Atualmente, muitas empresas já estão com um código repleto de bad smells, difícil de evoluir, de entender e de manter. O caos já está generalizado, e em algum momento a equipe já pensa em escrever tudo do zero. Nesse ponto é comum ocorrer uma rotatividade de profissionais na equipe, o que agrava ainda mais a situação. Um novo requisito que era para ser implementado e entregue em poucos dias, leva semanas. Bugs começam a aparecer sem explicação alguma. Um bug corrigido causa 10 novos bugs em outros lugares.

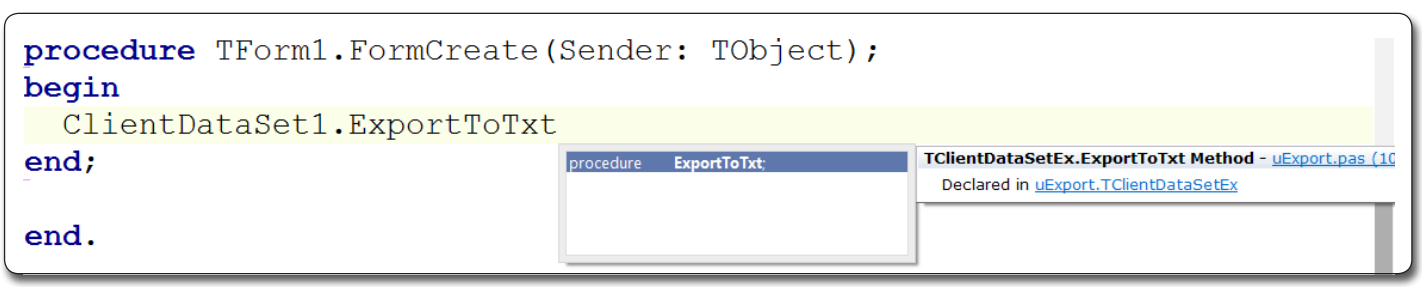


Figura 6. Método ExportToTxt agora aparece como membro de TClientDataSet

Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Listagem 11. Dando bons nomes para métodos

```
01 Errado:
02 // este método cadastra um cliente a partir do seu CPF
03 procedure CadCliCPF();
04
05 Certo:
06 procedure CadastrarClientePeloCPF();
```

Habitue-se a refatorar, buscar por bad smells e testar mais do que simplesmente programar, é uma prática não só saudável para o código-fonte, mas vital para o ciclo de vida da aplicação e da empresa. Algumas metodologias de desenvolvimento, inclusive, reforçam a importância dos testes e refatoração, como o TDD – *Test-Driven Development*, onde após desenvolver uma funcionalidade da forma mais simples possível para que ela seja aprovada nos testes, o programador deve voltar e refatorá-la, aumentando a qualidade do código. Uma outra prática bastante utilizada no desenvolvimento de sistemas atualmente é a revisão de código, onde um programador fica responsável por avaliar o código escrito por outro antes de integrá-lo ao repositório central do projeto. Em equipes que utilizam o Git e GitHub para o controle de versão, por exemplo, isso pode ser feito por meio de *pull requests*. Faça constantes auditorias no seu código, afinal, ele é o sangue que move sua empresa.

Referências Bibliográficas e Links:

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

[2] Fowler, M., "Refactoring: Improving the Design of Existing Programs", Addison-Wesley, 1999.

[3] Beck, K., "Test Driven Development: By Example", Addison-Wesley, 2002.

[4] Kerievsky, J., "Refactoring to Patterns", Addison-Wesley, 2008.

GoF Patterns – Quick Review
http://www.mydeveloperconnection.com/html/gof_design_patterns.htm

Refactoring - Martin Fowler
http://www.refactoring.com

Video – Débito de Projeto – Long Method
http://bit.ly/longmethod

Endereço do Code Central do Autor
https://downloads.embarcadero.com/Author/222668

Autor

Guinther Pauli
guinther.pauli@gmail.com

Consultor, Instrutor, Professor de Graduação e Pós-Graduação, Certificado oficial Microsoft, incluindo as credenciais MCP, MCAD, MCSD.NET, MCTS, MCPD (foco nas áreas ASP.NET, C#, MVC, Visual Studio e Arquitetura), Certificado oficial Borland / Embarcadero Delphi. MVP - Most Valuable Professional. Bacharel em Sistemas de Informação, MBA – Especialização em Gestão de Projetos e Mestre em Ciência da Computação.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

DEVMEDIA Copyright - Proibido copiar ou distribuir. Todos os direitos reservados para DevMedia

Guia HTML 5

Um verdadeiro manual de referência com tudo que você precisa sobre HTML!

DEVMEDIA

http://www.devmedia.com.br/guias/guia-html/3

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única. Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486